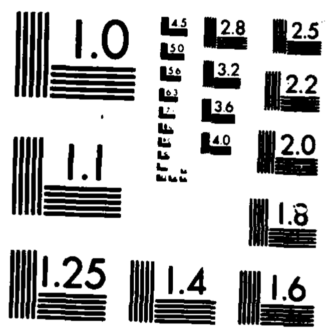
AD-A165 075 ADA (TRADEMARK) TRAINING CURRICULUM: ADVANCED ADA TOPICS L305 TEACHER'S GUIDE VOLUME 1(U) SOFTECH INC WALTHAM MA 1986 DAAB07-83-C-K506 1/5

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD-A165 075

1

1986

Ada® Training Curriculum

Advanced Ada® Topics
L305
Teacher's Guide
Volume I

Supersedes AD-A144498

86 3 11 143

U.S. Army Communications-Electronics Command
(CECOM)

Contract DAAB07-83-C-K506

Prepared By:

SOFTECH, INC.
460 Totten Pond Road
Waltham, MA 02154

PART I

BASIC STRUCTURING FEATURES

A

A-1

INSTRUCTOR NOTES

SECTION 1

PACKAGES AND NONSCALAR TYPES

INSTRUCTOR NOTES

- A PACKAGE SPECIFICATION IS A LIST OF DECLARATIONS TO BE USED OUTSIDE THE PACKAGE, I.E., IT DEFINES AN INTERFACE. THIS CONSISTS OF
  - CONSTANTS
  - VARIABLES
  - TYPE DECLARATIONS
  - PROCEDURES AND FUNCTIONS
  - EXCEPTIONS

  DO NOT DISCUSS PRIVATE PART.

- A PACKAGE BODY HAS THE SAME FORM AS A PROCEDURE BODY EXCEPT FOR THE FIRST LINE. THE DECLARATIONS ARE USED TO IMPLEMENT THE INTERFACE DEFINED BY THE PACKAGE SPECIFICATION. THE DECLARATIONS MUST AT LEAST INCLUDE SUBPROGRAM BODIES FOR ANY SUBPROGRAMS DEFINED BY THE INTERFACE. IN ADDITION, THE BODY MAY CONTAIN DECLARATIONS OF
  - CONSTANTS
  - VARIABLES
  - TYPE DECLARATIONS
  - PROCEDURES AND FUNCTIONS
  - EXCEPTIONS

  FOR USE WITHIN THE BODY, BUT <u>NOT</u> ACCESSIBLE OUTSIDE OF THE PACKAGE.

- MENTION THAT THE WORD BEGIN, THE STATEMENTS THAT FOLLOW IT, THE WORD EXCEPTION AND THE HANDLERS ARE OPTIONAL. THE EXCEPTION HANDLERS ONLY APPLY TO THE SEQUENCE OF INITIALIZATION STATEMENTS.

THE FORM OF A PACKAGE

A PACKAGE HAS TWO PARTS:

- PACKAGE SPECIFICATION -- DESCRIBES THE INTERFACE

```
package [package name] is
        [sequence of declarations]
end [package name] ;
```

- PACKAGE BODY -- DESCRIBES THE IMPLEMENTATION

```
package body [package name] is
        [sequence of declarations]
[begin -- [package name]
        [sequence of initialization statements]
         [exception
           [sequence of exception handlers]]]
end [package name] ;
```

INSTRUCTOR NOTES

- STUDENTS WHO HAVE TAKEN L202 HAVE SEEN THIS EXAMPLE BEFORE. HOWEVER, IT SO CLEARLY DISTINGUISHES BETWEEN INTERFACE AND IMPLEMENTATION THAT IT IS WORTH REPEATING.

- THE FRONT PANEL OF THE STEREO RECEIVER IS THE INTERFACE. THE ELECTRONIC COMPONENTS INSIDE THE RECEIVER ARE THE IMPLEMENTATION
  - THE INTERFACE EXPLAINS HOW THE ENTITIES PROVIDED BY A PACKAGE TO THE OUTSIDE ARE TO BE USED
  - THE IMPLEMENTATION EXPLAINS HOW THESE ENTITIES WORK INTERNALLY.

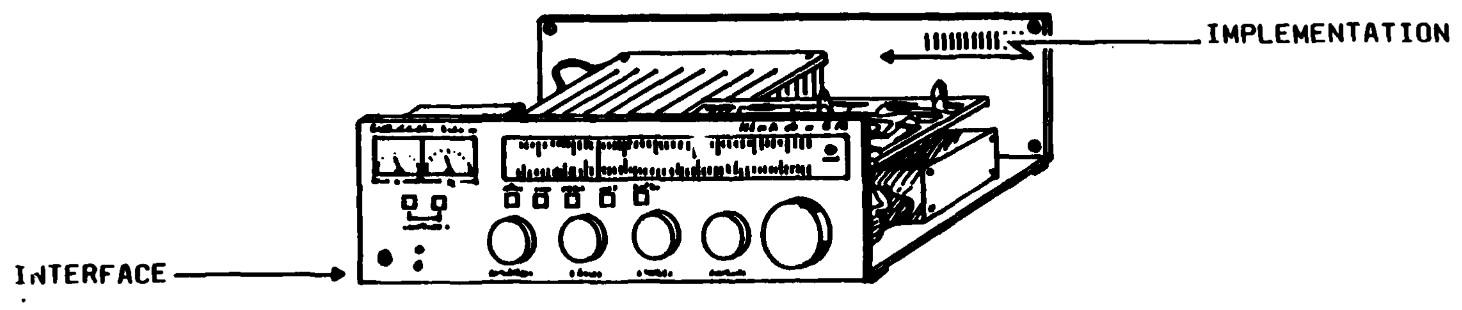
- MAKE SURE CLASS UNDERSTANDS THIS DISTINCTION. ANYONE WHO DOES NOT WILL HAVE MANY PROBLEMS LATER IN THE COURSE.

<u>INTERFACE</u> VERSUS <u>IMPLEMENTATION</u> OF A PACKAGE

INTERFACE: EXTERNAL APPEARANCE OF A PACKAGE

IMPLEMENTATION: INTERNAL WORKINGS OF A PACKAGE

ONLY THE INTERFACE IS RELEVANT TO THE USER OF A PACKAGE



INSTRUCTOR NOTES

- FOR THE INSTRUCTOR'S BENEFIT,

  Fibonacci (1) ≡ Fibonacci (2) ≡ 1

  Fibonacci (N) ≡ Fibonacci (N-2) + Fibonacci (N-1)

- THERE ARE FOUR ENTITIES IN THE INTERFACE : THE NAMED NUMBER Largest_Fibonacci_Number; THE SUBTYPE Fibonacci_Number_Subtype; THE FUNCTION Fibonacci; THE EXCEPTION Fibonacci_Error. THESE FOUR ENTITIES MAY BE NAMED OUTSIDE OF THE PACKAGE.

- THE PACKAGE BODY CONTAINS THE BODY FOR THE FUNCTION Fibonacci. THE BODY ALSO CONTAINS AN OBJECT DECLARATION FOR Fibonacci_Number_List. THIS ARRAY IS ONLY VISIBLE WITHIN THE PACKAGE BODY.

- NOTE THAT THE PACKAGE BODY ALSO CONTAINS A SEQUENCE OF INITIALIZATION STATEMENTS. THE AGGREGATE COULD HAVE BEEN USED IN THE DECLARATION OF Fibonacci_Number_List, THEREBY ELIMINATING THE NEED FOR INITIALIZATION STATEMENTS. IN THIS CASE WE WOULD ALSO NEED TO REMOVE BEGIN. ANOTHER APPROACH, FOR A LARGE RANGE OF FIBONACCI NUMBERS WOULD BE TO HAVE THE INITIALIZATION STATEMENTS

```
Fibonacci_Number_List (1) := 1;
Fibonacci_Number_List (2) := 1;
For N in 3 .. Fibonacci_Number_Subtype'Last loop
   Fibonacci_Number_List (N) := Fibonacci_Number_List (N-2) +
                                Fibonacci_Number_List (N-1);
end loop;
```

EXAMPLES

```
package Fibonacci_Package is
   Largest_Fibonacci_Number : constant := 16;
   subtype Fibonacci_Number_Subtype is Positive range 1 .. Largest_Fibonacci_Number;
   function Fibonacci (N : Fibonacci_Number_Subtype) return Positive;
      -- Yields Nth Fibonacci Number
   Fibonacci_Error : exception;
end Fibonacci_Package;

package body Fibonacci_Package is
   Fibonacci_Number_List : array (Fibonacci_Number_Subtype) of Positive;
   function Fibonacci (N : Fibonacci_Number_Subtype) return Positive is
   begin
      if N in Fibonacci_Number_Subtype then
         return Fibonacci_Number_List (N);
      else
         raise Fibonacci_Error;
      end if;
   end Fibonacci;

begin
   Fibonacci_Number_List := (1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987);
end Fibonacci_Package;
```

INSTRUCTOR NOTES

SELECTING A COMPONENT IN A PACKAGE SPECIFICATION

- INSIDE THE PACKAGE : BY ITS IDENTIFIER

```
Largest_Fibonacci_Number
Fibonacci_Number_Subtype
Fibonacci
```

- OUTSIDE THE PACKAGE : BY AN EXPANDED NAME

```
Fibonacci_Package.Largest_Fibonacci_Number
Fibonacci_Package.Fibonacci_Number_Subtype
Fibonacci_Package.Fibonacci
```

INSTRUCTOR NOTES

- IN GENERAL, EXPANDED NAMES PROVIDE DOCUMENTATION TO THE READER AS TO WHAT UNIT THE ENTITY IS DECLARED IN. USING A use CLAUSE REMOVES THIS DOCUMENTATION.

- use CLAUSE SHOULD NORMALLY ONLY BE USED WHEN PACKAGE PROVIDES OPERATOR SYMBOLS, E.G., IF A PACKAGE PROVIDES A "+" OPERATOR, WE WANT

```
Sum := X + Y;
```

  RATHER THAN

```
Sum := Package_Name."+" (Left => X, Right => Y);
```

  REMEMBER CLASS HAS NOT SEEN OVERLOADING OF OPERATOR SYMBOLS, SO SPEND ONLY ENOUGH TIME ON THIS TO EXPLAIN THE ABOVE EXAMPLE.

with CLAUSES AND use CLAUSES

- FOR SEPARATELY COMPILED PACKAGES, MUST USE A with CLAUSE FOR VISIBILITY

```
with Fibonacci_Package;
procedure Example is

   N : Fibonacci_Package.Fibonacci_Number_Subtype;
      ...
begin
   Fibonacci_Number := Fibonacci_Package.Fibonacci (N);
      ...
end Example;
```

- A use CLAUSE ELIMINATES THE NEED TO USE EXPANDED NAMES

```
with Fibonacci_Package; use Fibonacci_Package;
procedure Example is

   N : Fibonacci_Number_Subtype;
      ...
begin
   ...
   Fibonacci_Number := Fibonacci (N);
   ...
end Example;
```

INSTRUCTOR NOTES

THE FIRST FORM IS USED FOR RENAMING A VARIABLE OR A CONSTANT.

THE SECOND FORM IS USED FOR RENAMING AN EXCEPTION.

THE THIRD FORM IS USED FOR RENAMING A PACKAGE. IT IS USEFUL FOR PACKAGES NESTED INSIDE OTHER PACKAGES. (ONE OF THE RESOURCES PROVIDED BY A PACKAGE MAY ITSELF BE A PACKAGE.)

THE FOURTH FORM IS USED FOR RENAMING A PROCEDURE OR FUNCTION.

THERE ARE NO RENAMING DECLARATIONS FOR TYPES OR SUBTYPES, BUT A SUBTYPE DECLARATION CAN BE USED TO ACHIEVE THE EFFECT OF RENAMING A TYPE OR SUBTYPE.

THE NEW NAME MAY, BUT NEED NOT, BE IDENTICAL TO PART OF THE OLD NAME.

RENAMING DECLARATIONS

- PROVIDE A NEW NAME FOR SOME ENTITY, WHICH MAY BE MORE SUCCINCT OR MORE MEANINGFUL

- POSSIBLE FORMS

```
[IDENTIFIER] : [TYPE_MARK] renames [OBJECT_NAME];

e.g., Month_1:  Month_Type renames Date_1.Month_Type;


[IDENTIFIER] :  exception renames [EXCEPTION_NAME];

e.g., Singular_Matrix:  exception renames Matrix_Package.Singular_Matrix;


package [IDENTIFIER] renames [PACKAGE_NAME];

e.g., package Keypad_Interface renames Hardware_Interface.Keypad_Interface;


[SUBPROGRAM_SPECIFICATION] renames [SUBPROGRAM_NAME];

e.g., function Factorial (N:  Argument_Subtype) return Result_Type
          renames Factorial_Package.Factorial;


subtype [IDENTIFIER] is [SUBTYPE_INDICATION];

e.g., subtype Factorial_Argument_Subtype is Factorial_Package.Argument_Subtype;
```

INSTRUCTOR NOTES

THIS SECTION ON ARRAYS IS A REVIEW OF MATERIAL COVERED IN L202.

TECHNICALLY, A "CONSTRAINED ARRAY TYPE" IS REALLY A CONSTRAINED SUBTYPE OF AN ANONYMOUS UNCONSTRAINED ARRAY TYPE. THIS SECTION IGNORES THIS DETAIL.

ARRAY TYPES

- A VALUE IN AN ARRAY TYPE IS A COLLECTION OF COMPONENTS.
  - EACH COMPONENT IS IDENTIFIED BY A UNIQUE COMBINATION OF INDEX VALUES.
  - ALL COMPONENTS BELONG TO THE SAME SUBTYPE

- AN ARRAY TYPE IS CHARACTERIZED BY:
  - THE NUMBER OF INDEX VALUES USED TO IDENTIFY THE COMPONENTS (THE NUMBER OF INDEX POSITIONS OR DIMENSIONS)
  - FOR EACH DIMENSION, THE SUBTYPE OF THE INDEX VALUES IN THAT DIMENSION.
  - THE SUBTYPE OF THE COMPONENTS
  - WHETHER THE ARRAY TYPE IS CONSTRAINED OR UNCONSTRAINED
    - CONSTRAINED: ALL OBJECTS IN THE ARRAY TYPE HAVE THE SAME RANGE OF INDEX VALUES IN A GIVEN DIMENSION
    - UNCONSTRAINED: DIFFERENT OBJECTS IN THE ARRAY TYPE MAY HAVE DIFFERENT RANGES OF INDEX VALUES IN A GIVEN DIMENSION

INSTRUCTOR NOTES

BULLET 1: AN "INDEX SUBTYPE DESCRIPTION" OF ONE OF THESE FORMS IS CALLED A <u>DISCRETE RANGE</u>. DISCRETE RANGES ALSO OCCUR IN INDEX CONSTRAINTS, FOR LOOPS, SLICES, AND THE CHOICE LISTS OF CASE STATEMENTS, AGGREGATES, AND VARIANT PARTS.

ITEM 3: THE TYPE INTEGER IS ALSO ASSUMED IF ONE OR BOTH BOUNDS ARE ATTRIBUTES WITH UNIVERSAL INTEGER VALUES. WHEN TYPE INTEGER IS NOT ASSUMED, THERE MUST BE AT MOST ONE TYPE THAT BOTH EXPRESSIONS MIGHT BELONG TO.

IF THERE IS MORE THAN ONE INTEGER TYPE, THE DISCRETE RANGE -10 .. 10 IS ILLEGAL, BUT THE FOLLOWING DISCRETE RANGES ARE LEGAL (ASSUMING Negative_Ten IS A NAMED NUMBER EQUAL TO -10):

```
Negative_Ten .. 10
Integer range  -10 .. 10
```

BULLET 2: THE SYMBOL <> IS PRONOUNCED "BOX" AND CONNOTES INFORMATION TO BE SUPPLIED LATER.

ARRAY TYPE DECLARATIONS

```
type [array type name] is
   array ([index subtype description] {, [index subtype description]})
      of [component subtype description] ;
```

- FORMS OF [index subtype description] FOR CONSTRAINED ARRAY TYPES:
  - [discrete type mark] range [lower bound] .. [upper bound]
  - [discrete type mark]
  - [lower bound] .. [upper bound] (type Integer assumed if both bounds are integer literals or named numbers)

- FORM OF [index subtype description] FOR UNCONSTRAINED ARRAY TYPES

  [discrete type mark] range <>

- A [discrete type mark] IS AN IDENTIFIER NAMING AN INTEGER TYPE, AN ENUMERATION TYPE, OR A SUBTYPE OF SUCH A TYPE.

INSTRUCTOR NOTES

THE NEXT SLIDE ILLUSTRATES THESE RULES.

INDEX CONSTRAINTS

- <u>FORM</u>: (index subtype description (, index subtype description)) WHERE EACH index subtype description IS OF ONE OF THE FORMS ALLOWED IN CONSTRAINED ARRAY TYPE DECLARATIONS.

- <u>PLACEMENT</u>: FOLLOWING THE NAME OF AN UNCONSTRAINED ARRAY TYPE OR SUBTYPE, TO SPECIFY SPECIFIC INDEX BOUNDS IN EACH DIMENSION.

- AN ARRAY OBJECT DECLARATION MUST SPECIFY FIXED BOUNDS FOR EACH DIMENSION.
  - IT MAY SPECIFY A CONSTRAINED ARRAY TYPE.
  - IT MAY SPECIFY AN *UNCONSTRAINED ARRAY TYPE PLUS* AN INDEX CONSTRAINT.

- FIXED INDEX BOUNDS MUST ALSO BE SPECIFIED WHEN ARRAYS ARE USED AS COMPONENTS OF OBJECTS (IN ARRAYS OF ARRAYS OR IN RECORDS CONTAINING ARRAYS)

- FOLLOWING THE SUBTYPE DECLARATION

```
subtype [subtype name] is [unconstrained array type name] [index constraint];
```

THE SUBTYPE NAME MAY BE USED TO STAND FOR THE TYPE NAME FOLLOWED BY THE INDEX CONSTRAINT.

<u>LEGAL EXAMPLES</u>:

A IS DECLARED TO BE AN OBJECT IN A CONSTRAINED ARRAY TYPE.

B IS DECLARED TO BE AN OBJECT IN AN UNCONSTRAINED ARRAY TYPE, WITH BOUNDS SPECIFIED BY AN INDEX CONSTRAINT.

C IS DECLARED TO BE AN OBJECT OF SUBTYPE List_of_10_Subtype, WHICH IS EQUIVALENT TO Integer_List_Type (1 .. 10).

Bowling_Scoresheet_Type IS AN ARRAY TYPE WHOSE COMPONENT TYPE IS A CONSTRAINED ARRAY SUBTYPE. (THE FACT THAT Bowling_Scoresheet_Type ITSELF IS UNCONSTRAINED IS IRRELEVANT.)

Employee_Type IS A RECORD TYPE WITH A COMPONENT BELONGING TO THE CONSTRAINED ARRAY TYPE Time_Card_Type.

<u>ILLEGAL EXAMPLES</u>

D: ATTEMPT TO DECLARE AN OBJECT IN AN UNCONSTRAINED ARRAY TYPE WITHOUT AN INDEX CONSTRAINT.

Set_List_Type: ATTEMPT TO USE AN UNCONSTRAINED ARRAY TYPE AS THE COMPONENT TYPE OF ANOTHER ARRAY TYPE. (THE FACT THAT Set_List_Type ITSELF WOULD BE CONSTRAINED IS IRRELEVANT.)

Department_Type: ATTEMPT TO USE AN UNCONSTRAINED ARRAY TYPE AS A RECORD COMPONENT TYPE.

EXAMPLES

```
type Day_Type is (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday);
type Time_Card_Type is array (Day_Type) of Float;
type Integer_List_Type is array (Positive range <>) of Integer;
subtype List_of_10_Subtype is Integer_List_Type (1 .. 10);
```

LEGAL DECLARATIONS:

```
A: Time_Card_Type;
B: Integer_List_Type (Integer range 1 .. 10);
C: List_of_10_Subtype;
type Bowling_Scoresheet_Type is array (Positive range <>) of List_of_10_Subtype;
type Employee_Type is
   record
      Employee_Number : Positive;
      Employee_Hours  : Time_Card_Type;
   end record;
```

WHY ARE THE FOLLOWING DECLARATIONS ILLEGAL?

```
D: Integer_List_Type;
type Set_List_Type is array (1 .. 10) of Integer_List_Type;
type Department_Type is
   record
      Department_Number    : Positive;
      Employee_Number_List : Integer_List_Type;
   end record;
```

INSTRUCTOR NOTES

A 'Range ATTRIBUTE MAY BE USED AS A DISCRETE RANGE.

ATTRIBUTES OF ARRAYS AND ARRAY TYPES

- IN EACH CASE, A MAY BE EITHER AN ARRAY OBJECT, A CONSTRAINED ARRAY TYPE, OR A CONSTRAINED ARRAY SUBTYPE. IT MAY NOT BE AN UNCONSTRAINED ARRAY TYPE OR SUBTYPE:

A'First (dimension)	LOWER INDEX BOUND
A'Last (dimension)	UPPER INDEX BOUND
A'Length (dimension)	NUMBER OF INDEX VALUES
A'Range (dimension)	STANDS FOR THE RANGE A'First (dimension) .. A'Last (dimension)

- THE dimension MUST BE A STATIC EXPRESSION (GENERALLY AN INTEGER LITERAL).

- THE (dimension) MAY BE OMITTED, IN WHICH CASE DIMENSION (1) IS ASSUMED
  - THIS PRACTICE IS RECOMMENDED FOR ONE-DIMENSIONAL ARRAYS
  - IT IS NOT RECOMMENDED FOR MULTI-DIMENSIONAL ARRAYS

INSTRUCTOR NOTES

Average_of_Nothing_Error IS RAISED WHEN Average_Value IS CALLED WITH A NULL ARRAY.

THE AVERAGE IS UNDEFINED IN SUCH A CASE.

SUBPROGRAMS MANIPULATING ARRAYS

- A SUBPROGRAM PARAMETER TYPE OR RESULT TYPE MAY BE AN UNCONSTRAINED ARRAY TYPE. DURING A GIVEN CALL, THE FORMAL PARAMETER ASSUMES THE CONSTRAINTS OF THE ACTUAL PARAMETER.

- THIS ALLOWS THE WRITING OF GENERAL-PURPOSE SUBPROGRAMS TO HANDLE ARRAYS OF DIFFERENT SIZES.

- CONTEXT FOR EXAMPLE:

```
type Data_List_Type is array (Positive range<>) of Float;
Average_Of_Nothing_Error : exception;
```

- EXAMPLE:

```
function Average_Value (List : Data_List_Type) return Float is
   Sum : Float := 0.0;
begin -- Average_Value
   if List'Length = 0 then
      raise Average_Of_Nothing_Error;
   else
      for I in List'Range loop
         Sum := Sum + List (I);
      end loop;
      return Sum/Float (List'Length);
   end if;
end Average_Value;
```

INSTRUCTOR NOTES

THE ITEM BETWEEN THE PARENTHESES IN A SLICE MAY BE ANY DISCRETE RANGE.

SLICES AND CATENATION

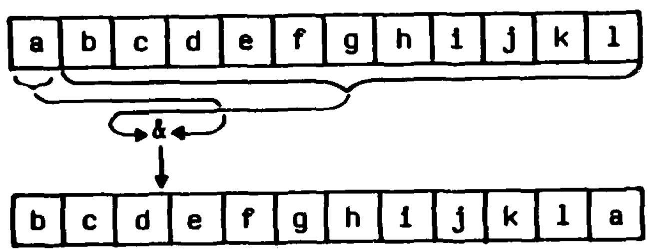
- IF A IS ONE-DIMENSIONAL ARRAY, A(i .. j) IS A ONE-DIMENSIONAL ARRAY CONSISTING OF THE CONSECUTIVE COMPONENTS OF A CORRESPONDING TO INDEX VALUES i THROUGH j.

- A(i .. j) IS CALLED A SLICE.

- THE OPERATOR & TAKES TWO ONE-DIMENSIONAL ARRAYS, AN ARRAY AND A COMPONENT, OR TWO COMPONENTS AND STICKS THEM TOGETHER TO FORM A LONGER ARRAY. THIS IS CALLED CATENATION.

- THE FOLLOWING ASSIGNMENT SHIFTS THE COMPONENTS OF A LEFTWARD, ROTATING THE LEFTMOST COMPONENT INTO THE RIGHTMOST POSITION:

```
A := A(A'First + 1 .. A'Last) & A(A'First);
```



INSTRUCTOR NOTES

BULLET 2: THERE ARE MANY RESTRICTIONS ON THE USE OF others IN AN ARRAY AGGREGATE. IT MAY ONLY BE USED IN CERTAIN CONTEXTS WHERE THE INDEX BOUNDS OF THE ARRAY ARE KNOWN. (SEE RM 4.3.2).

BULLET 3: DESPITE THIS, THE TWO KINDS OF ARRAYS ARE NOT EQUIVALENT IN Ada (AS THEY ARE IN Pascal). FOR AN ARRAY A OF THE FIRST TYPE, INDEX COMPONENTS A(i1), A(i1)(i2), AND A(i1)(i2)(i3) ARE ALL LEGAL EXPRESSIONS. FOR AN ARRAY B OF THE SECOND TYPE, THE ONLY LEGAL FORM OF INDEX COMPONENT IS A(i1, i2, i3).

AGGREGATES

- AGGREGATES ARE COMPONENT-BY-COMPONENT DESCRIPTIONS OF VALUES IN ARRAY TYPES.

- SEVERAL FORMS FOR ONE-DIMENSIONAL ARRAYS:

  - COMPLETE ORDERED LIST OF VALUES:

    ```
    (3, 7, 0, 7, 6, 3, 0, 0, 0, 0)
    ```

  - PARTIAL ORDERED LIST OF VALUES:

    ```
    (3, 7, 0, 7, 6, 3, others => 0)
    ```

  - NON-ORDERED LIST OF VALUES IDENTIFIED BY INDICES:

    ```
    (1| 6 => 3, 2| 4 => 7, 3 => 0, 5 => 6, 7 .. 10 => 0)
    (2| 4 => 7, 1| 6 => 3, 3 => 0, 5 => 6, others => 0)
    (1| 6 => 3, 2| 4 => 7, 3| 7 .. 10 => 0, 5 => 6)
    ```

- AN AGGREGATE FOR AN ARRAY (Type_1, Type_2, Type_3) OF Type_4 IS WRITTEN AS IF IT WERE AN AGGREGATE FOR AN ARRAY (Type_1) OF ARRAY (Type_2) OF ARRAY (Type_3) OF Type_4, WITH AGGREGATES NESTED INSIDE AGGREGATES.

    ```
    (1 => (A => (TRUE, TRUE),  B => (TRUE, FALSE)),
     2 => (A => (FALSE, TRUE), B => (FALSE, FALSE)))
    ```

INSTRUCTOR NOTES

THE ASSIGNMENT IS ILLEGAL BECAUSE A DECLARATION LIKE

```
A, B, C:  array (1 .. 10) of Integer;
```

IS EQUIVALENT TO A SEQUENCE OF INDIVIDUAL DECLARATIONS:

```
A: array (1 .. 10) of Integer;
B: array (1 .. 10) of Integer;
C: array (1 .. 10) of Integer;
```

THUS A, B, AND C BELONG TO THREE DIFFERENT ANONYMOUS ARRAY TYPES.

A SHORTHAND FOR ONE-OF-A-KIND ARRAYS

- INSTEAD OF

```
type Table_Type is array (1 .. 10) of Integer;
Table : Table_Type;
```

  YOU MAY WRITE

```
Table : array (1 .. 10) of Integer;
```

- THE TYPE OF Table IS THEN ANONYMOUS. NO OTHER OBJECT OR SUBPROGRAM PARAMETER MAY EVER BELONG TO THE SAME TYPE AS TABLE.

- THIS IS ONLY ALLOWED FOR OBJECT DECLARATIONS, NOT RECORD COMPONENT DECLARATIONS.

- SIMILAR SHORTHANDS ARE NOT PERMITTED FOR OTHER KINDS OF TYPES.

- IN THE FOLLOWING IS THE ASSIGNMENT LEGAL?

```
Data_List_One, Data_List_Two : array (1 .. 20) of Float;
  ...
Data_List_One := Data_List_Two;
```

INSTRUCTOR NOTES

RECORD TYPES

INSTRUCTOR NOTES

VALUES THAT ARE MEANT TO BE UNDERSTOOD IN CONJUNCTION WITH EACH OTHER ARE GOOD CANDIDATES FOR INCLUSION IN A RECORD TYPE.

THE Longitude_Type DECLARATION SHOWS:

- COMPONENT DECLARATIONS MAY INCLUDE CONSTRAINTS.
- COMPONENT DECLARATIONS MAY BE COMBINED INTO A SINGLE COMPONENT DECLARATION.
- COMPONENTS MAY BE GIVEN DEFAULT INITIAL VALUES.

THE OBJECT DECLARATIONS SHOW:

- VARIABLES OR CONSTANTS MAY BE DECLARED IN A RECORD TYPE.
- OBJECTS MAY BE GIVEN INITIAL VALUES. (THESE OVERRIDE THE DEFAULT INITIAL VALUES FOR INDIVIDUAL COMPONENTS GIVEN IN A TYPE DECLARATION.)
- RECORD TYPE VALUES MAY BE DESCRIBED BY AGGREGATES (ADDRESSED ON THE NEXT SLIDE).

THE FIRST ASSIGNMENT SHOWS THAT RECORD-TYPE VALUES MAY BE TREATED AS MONOLITHIC ENTITIES. IN PARTICULAR, AN ENTIRE RECORD-TYPE VALUE CAN BE ASSIGNED TO A RECORD-TYPE VARIABLE.

THE NESTED if STATEMENTS INCREASE Current_Longitude BY ONE SECOND, ACCOUNTING FOR WRAP-AROUND OF SECONDS, MINUTES, OR DEGREES. THE NESTED if STATEMENTS SHOW:

- RECORD COMPONENTS MAY BE NAMED INDIVIDUALLY BY NAMES OF THE FORM $\boxed{\text{OBJECT NAME}}\,.\,\boxed{\text{COMPONENT NAME}}$
- SUCH NAMES MAY APPEAR IN EXPRESSIONS OR, IF THE OBJECT NAMED BEFORE THE DOT IS A VARIABLE, AS VARIABLES (E.G. ON THE LEFTHAND SIDE OF AN ASSIGNMENT STATEMENT).

SIMPLE RECORD TYPES

VALUES IN A RECORD TYPE ARE COMBINATIONS OF VALUES IN OTHER TYPES.

TAKEN TOGETHER, THE COMPONENT VALUES OF A RECORD TYPE VALUE DESCRIBE A SINGLE ABSTRACT NOTION.

```
type Longitude_Type is
   record
      Degrees_Part                  : Integer range -180 .. 180;
      Minutes_Part, Seconds_Part : Integer range 0 .. 59 := 0;
   end record;

Current_Longitude, Previous_Longitude : Longitude_Type;
Prime_Meridian                        : constant Longitude_Type := (0, 0, 0);
   ...

Previous_Longitude := Current_Longitude;

if Current_Longitude.Seconds_Part = 59 then
   if Current_Longitude.Minutes_Part = 59 then
      if Current_Longitude.Degrees_Part = 179 then
         Current_Longitude.Degrees_Part := -180;
      else
         Current_Longitude.Degrees_Part := Current_Longitude.Degrees_Part + 1;
      end if;
      Current_Longitude.Minutes_Part := 0;
   else
      Current_Longitude.Minutes_Part = Current_Longitude.Minutes_Part + 1;
   end if;
   Current _Longitude.Seconds_Part := 0;
else
   Current_Longitude.Seconds_Part := Current_Longitude.Seconds_Part + 1;
end if;
```

INSTRUCTOR NOTES

ALL AGGREGATES ON THIS SLIDE ARE EQUIVALENT.

IN A POSITIONAL AGGREGATE, VALUES ARE ASSOCIATED WITH COMPONENTS BASED ON THE ORDER IN WHICH THEY APPEAR.

IN A NAMED AGGREGATE, VALUES ARE ASSOCIATED WITH COMPONENTS BASED ON COMPONENT NAMES GIVEN IN THE AGGREGATE. COMPONENTS CAN BE NAMED IN ANY ORDER. THE VERTICAL BAR NOTATION ALLOWS SEVERAL COMPONENTS WITH THE SAME TYPE TO BE GIVEN THE SAME VALUE IN A SUCCINCT WAY. THE <u>others</u> NOTATION IS RARE IN RECORD AGGREGATES. IT APPLIES WHEN ALL REMAINING COMPONENTS BELONG TO THE SAME TYPE AND ARE TO BE GIVEN THE SAME VALUE.

<u>UNLIKE ARRAY AGGREGATES</u>, RECORD AGGREGATES MAY CONTAIN A COMBINATION OF POSITIONAL AND NAMED NOTATION. THE POSITIONAL PART MUST COME FIRST.

REVIEW OF RECORD AGGREGATES

CONTEXT:

```
type Longitude_Type is
   record
      Degrees_Part                 : Integer range -180 .. 180;
      Minutes_Part, Seconds_Part : Integer range 0 .. 59 := 0;
   end record;
```

<u>POSITIONAL</u>

```
(90, 0, 0)
```

<u>NAMED</u>

```
(Minutes_Part | Seconds_Part => 0, Degrees_Part => 90)
(Degrees_Part => 90, others => 0)
```

<u>COMBINATION</u>

```
(90, Seconds_Part => 0, Minutes_Part => 0)
(90, others => 0)
```

INSTRUCTOR NOTES

A PREVIOUS VERSION OF Ada ALLOWED RECORD TYPE COMPONENTS TO BELONG TO ANONYMOUS ARRAY TYPES, E.G.:

```
type Flight_Plan_Type is
   record
      Number_Of_Points_Part : Integer range 0 .. 10 := 0;
      Point_List_Part       : array (1 .. 10) of Location_Type;
   end record;
```

THE STANDARD NOW IN EFFECT (FEBRUARY 1983) DOES NOT ALLOW THIS.

THE THREE POSSIBLE COMBINATIONS OF COMPOSITE TYPES ARE ILLUSTRATED ON THE NEXT SLIDE.

COMBINATION OF RECORD TYPES WITH OTHER TYPES

- RECORD TYPES MAY BE USED FOR COMPONENTS OF OTHER RECORD TYPES.
- RECORD TYPES MAY BE USED FOR COMPONENTS OF ARRAY TYPES.
- ARRAY TYPES MAY BE USED FOR COMPONENTS OF RECORD TYPES, BUT:
  - THE ARRAY TYPE MAY NOT BE ANONYMOUS. (IT MUST HAVE BEEN DECLARED IN A PREVIOUS ARRAY TYPE DECLARATION.)
  - THE ARRAY-VALUED RECORD COMPONENT MUST BE CONSTRAINED. (THE RECORD TYPE DECLARATION GIVES EITHER THE NAME OF A CONSTRAINED ARRAY SUBTYPE OR THE NAME OF AN UNCONSTRAINED ARRAY SUBTYPE PLUS AN INDEX CONSTRAINT.)

INSTRUCTOR NOTES

PARSE EACH OF THE SUBCOMPONENT NAMES ALOUD, EXPLAINING THE MEANING OF EACH NAME.

USING ONE TYPE TO DEFINE ANOTHER ALLOWS ONE TO VIEW DATA AT DIFFERENT <u>LEVELS OF ABSTRACTION</u>. ONE CAN THINK OF A FLIGHT PLAN AS CONTAINING A COUNT OF <u>LOCATIONS</u> AND A <u>LIST OF LOCATIONS</u> WITHOUT WORRYING ABOUT HOW A LOCATION IS REPRESENTED. AT A LOWER LEVEL OF ABSTRACTION, ONE CAN THINK OF A LOCATION AS CONSISTING OF A LATITUDE AND LONGITUDE WITHOUT WORRYING ABOUT HOW LATITUDES AND LONGITUDES ARE REPRESENTED. AT A STILL LOWER LEVEL, ONE CAN THINK OF A LONGITUDE AS CONSISTING OF DEGREES, MINUTES, AND SECONDS.

NAMES AS COMPLEX AS THE LAST ONE ARE A SYMPTOM THAT THE PROGRAMMER IS WORKING AT TOO MANY LEVELS OF ABSTRACTION SIMULTANEOUSLY.

IT MAY BE MORE APPROPRIATE, FOR EXAMPLE TO CALL A SUBPROGRAM WITH THE ACTUAL PARAMETER Flight_Plan.Point_List_Part (N) CORRESPONDING TO THE FORMAL PARAMETER Point, AND TO REFER TO Point.Longitude_Part.Minutes_Part INSIDE THE SUBPROGRAM.

IT MAY BE HELPFUL FOR THE INSTRUCTOR TO DRAW THE DATA STRUCTURE ON THE BOARD IF THE CLASS HAS TROUBLE FOLLOWING IT.

THE DECLARATION FOR Latitude_Type IS SIMILAR TO THAT FOR Longitude_Type AND IS THEREFORE NOT INCLUDED ON THE FOIL.

NAMES OF SUBCOMPONENTS

TYPE DECLARATIONS:

```
type Longitude_Type is
   record
      Degrees_Part                 : Integer range -180 .. 180;
      Minutes_Part, Seconds_Part : Integer range 0 .. 59 := 0;
   end record;

type Location_Type is
   record
      Longitude_Part : Longitude_Type;
      Latitude_Part  : Latitude_Type;
   end record;

type Location_List_Type is array (1 .. 10) of Location_Type;

type Flight_Plan_Type is
   record
      Number_Of_Points_Part : Integer range 0 .. 10 := 0;
      Point_List_Part       : Location_List_Type;
   end record;
```

VARIABLE DECLARATIONS:

```
Location      : Location_Type;
Location_List : Location_List_Type;
Flight_Plan   : Flight_Plan_Type;
N             : Integer range 1 .. 10;
```

SOME SUBCOMPONENTS:

```
Location.Longitude_Part.Minutes_Part (record component of a record component)
Location_List (N).Longitude_Part     (record component of an array component)
Flight_Plan.Point_List_Part (N)      (array component of a record component)

Flight_Plan.Point_List_Part (N).Longitude_Part.Minutes_Part (inappropriately complex)
```

INSTRUCTOR NOTES

- DISCRIMINANTS ARE RECORD COMPONENTS THAT ACT AS PARAMETERS OF THE RECORD TYPE.

- OBJECTS IN Buffer_Type NEED NOT BE GIVEN EXPLICIT INITIAL VALUES WHEN DECLARED, BUT OBJECTS IN Varying_String_Type MUST ALWAYS BE. THIS IS COVERED IN A LATER SLIDE.

DISCRIMINANTS ALLOW CONTROL OVER ARRAY SIZE

- EXAMPLE

```
type Element_List_Type is array (Positive range <>) of Element_Type;
subtype Priority_Subtype is Positive range 1 .. 10;

type Buffer_Type (Size_Part : Positive := 5; Priority_Part : Priority_Subtype := 1) is
   record
      Element_List_Part : Element_List_Type (1 .. Size_Part);
      Length_Part       : Natural := 0;
   end record;
```

- EXAMPLE

```
Type Varying_String_Type (Size_Part : Natural) is
   record
      Contents_Part : String (1 .. Size_Part);
      Length_Part   : Natural := 0;
   end record;
```

- DECLARATIONS

```
Standard_Priority_Buffer : Buffer_Type (Priority_Part => 5, Size_Part => 70);
High_Priority_Buffer     : Buffer_Type (20, Priority_Part => Priority_Subtype'Last);
Low_Priority_Buffer      : Buffer_Type;

Message : Varying_String_Type (Size_Part => 80);
```

INSTRUCTOR NOTES

DISCRIMINANTS ALLOW RECORD TYPES TO HAVE DIFFERENT FORMS

- EXAMPLE

```
type Table_Type (Size_Part : Positive := 100;
                 Collecting_Statistics : Boolean := False) is
   record
      Element_List_Part : Element_List_Type (1 .. Size_Part);
      case Collecting_Statistics is
         when False =>
            null;
         when True =>
            Number_Of_Look_Ups_Part   : Natural := 0;
            Number_Of_Successes_Part : Natural := 0;
      end case;
   end record;
```

- DECLARATIONS

```
Collecting_User_Statistics : constant Boolean := True;

User_Table : Table_Type (Size_Part => Requested_Size,
                         Collecting_Statistics =>  Collecting_User_Statistics);
   -- THIS Table_Type OBJECT HAS COMPONENTS
         Number_Of_Look_Ups_Part AND Number_Of_Successes_Part

System_Table : Table_Type;
   --  THIS Table_Type OBJECT DOES NOT
```

INSTRUCTOR NOTES

- MENTION THAT $\boxed{\text{CHOICE\_LIST}}$ HAS THE SAME FORM IN A CASE STATEMENT

  $\boxed{\text{CHOICE}}\ \{\ |\ \boxed{\text{CHOICE}}\ \}$

  WITH EACH $\boxed{\text{CHOICE}}$ AN EXPRESSION OR A RANGE.

GENERAL FORM

```
type [TYPE NAME] ([DISCRIMINANT_SPECIFICATION] {; [DISCRIMINANT_SPECIFICATION]}) is
   record
      [COMPONENT LIST]
   end record;


WHERE A [DISCRIMINANT_SPECIFICATION] is
   [IDENTIFIER] {, [IDENTIFIER]} : [DISCRETE_TYPE_NAME] [:= [DEFAULT_INITIAL_VALUE]]
AND A [COMPONENT_LIST] IS ONE OF
   [ORDINARY_COMPONENT_DECLARATION]
  {[ORDINARY_COMPONENT_DECLARATION]}    |    [VARIANT_PART]    |    null;
  [[VARIANT_PART]]

AND WHERE A [VARIANT_PART] IS

   case [DISCRIMINANT_NAME] is
      when [CHOICE_LIST] =>
           [COMPONENT_LIST]
      when [CHOICE_LIST] =>
           [COMPONENT_LIST]
   end case;
```

INSTRUCTOR NOTES

- THIS SLIDE EXPLAINS WHEN DISCRIMINANT CONSTRAINTS ARE NEEDED.

- THE NEXT SLIDE LOOKS AT CHANGING DISCRIMINANT VALUES.

DISCRIMINANT CONSTRAINTS AND DEFAULT INITIAL VALUES

- A RECORD TYPE DECLARATION MUST PROVIDE DEFAULT INITIAL VALUES FOR <u>ALL</u> OF ITS DISCRIMINANTS OR FOR NONE

```
type Buffer_Type (Size_Part : Positive := 5; Priority_Part : Priority_Subtype) is
                                                   **ILLEGAL**
```

- IF DEFAULT INITIAL VALUES ARE PROVIDED, AN OBJECT IN THE TYPE NEED NOT BE DECLARED WITH A DISCRIMINANT CONSTRAINT

```
Low_Priority_Buffer  : Buffer_Type;
High_Priority_Buffer : Buffer_Type (Size_Part => 20,
                                    Priority_Part => Priority_Subtype'Last);
```

OTHERWISE, A DISCRIMINANT CONSTRAINT MUST ALWAYS BE SPECIFIED

```
Line    : Varying_String_Type (Size_Part => 80);  -- LEGAL

Message : Varying_String_Type;                     -- ILLEGAL
```

- AN OBJECT DECLARED WITH A DISCRIMINANT CONSTRAINT IS <u>CONSTRAINED</u>, I.E., DISCRIMINANTS NEVER CHANGE:

    High_Priority_Buffer, Line

  AN OBJECT DECLARED WITHOUT A DISCRIMINANT CONSTRAINT IS <u>UNCONSTRAINED</u>, I.E., DISCRIMINANTS MAY CHANGE:

    Low_Priority_Buffer

INSTRUCTOR NOTES

- BULLET 1 DISCRIMINANT DETERMINES WHAT OTHER DATA THE RECORD MAY CONTAIN.

  CHANGING <u>ONLY</u> THE DISCRIMINANT COULD LEAD TO AN INCONSISTENT STATE.

USING DISCRIMINANTS

- ILLEGAL TO CHANGE A DISCRIMINANT COMPONENT OF A RECORD BY ITSELF
  - NOT THROUGH ASSIGNMENT (:=)
  - NOT THROUGH SUBPROGRAM CALLS

- FOR UNCONSTRAINED RECORD OBJECTS MAY ASSIGN AN <u>ENTIRE</u> RECORD VALUE WITH DIFFERENT DISCRIMINANT COMPONENTS
  - IF THE DISCRIMINANT SELECTS A VARIANT, AN AGGREGATE FOR THE RECORD TYPE MAY NOT HAVE VARIABLES OR FUNCTION CALLS IN THE COMPONENT FOR THE DISCRIMINANT

INSTRUCTOR NOTES

ACCESS TYPES

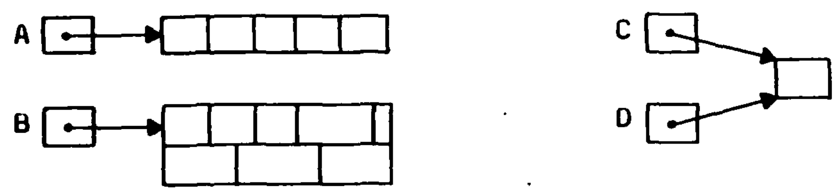
INSTRUCTOR NOTES

- THIS IS A QUICK REVIEW. MORE ON ALLOCATING OBJECTS TO FOLLOW. THIS SLIDE SHOULD BE USED TO REVIEW OPERATIONS AND TERMINOLOGY.

ACCESS DATA TYPES

DECLARATION : type [access type name] is access [designated subtype] [[constraint]];

CONTEXT:



OPERATIONS	EXAMPLE	
1. DYNAMIC ALLOCATION	A := new Array_Type;	
	B := new Record_Type;	
	C := new Integer;	
2. ASSIGNMENT	C := D	
3. DEREFERENCING	A (3)	B.component
	A (3 .. 5)	B.all
	A'Length	C.all + 1
4. RELATIONAL	-- DO C AND D POINT TO THE SAME ALLOCATOR?	
	IF C = D THEN ...	

INSTRUCTOR NOTES

```
ANSWER FOR A (3 .. 5) is A.all (3 .. 5)
ANSWER FOR A'First    is A.all'First
```

REVIEW DEREFERENCING

REMEMBER ...

A (4) IS A SHORTHAND FOR A.all (4)

A (3 .. 5) is a shorthand for ______________________.

A'First is a shorthand for _________________________.

IF A NAMES THE ACCESS VALUES,

A.all NAMES THE VARIABLE POINTED TO BY THE ACCESS VALUE

INSTRUCTOR NOTES

- THIS IS THE FIRST OF FOUR SLIDES REVIEWING ALLOCATORS

- EXAMPLE 1
  - THIS SHOWS THE ALLOCATION OF A CONSTRAINED ARRAY OBJECT
  - IN THIS EXAMPLE, AND THE REST, NOTE THAT THE TYPE NAME OF THE OBJECTS BEING POINTED TO IS USED IN THE ALLOCATOR

- EXAMPLE 2
  - THIS SHOWS THE ALLOCATION OF AN OBJECT IN A RECORD TYPE WITHOUT DISCRIMINANTS
  - WHEN ALLOCATED, THE OBJECT POINTED TO BY Flight_Plan_Pointer WILL HAVE ITS Number_Of_Points_Part INITIALIZED TO ZERO, JUST AS FOR DECLARED OBJECTS

ALLOCATING OBJECTS - FIRST FORM

new $\boxed{\text{TYPE MARK}}$

- EXAMPLE 1

```
subtype Name_Type is String (1 .. 32);
type Name_Pointer_Type is access Name_Type;
   ...
Name : Name_Type;
Name_Pointer : Name_Pointer_Type;
   ...
Name_Pointer := new Name_Type;
Name_Pointer.all := Name;
```

- EXAMPLE 2

```
   ...
type Flight_Plan_Type is
   record
      Number_Of_Points_Part : Integer range 0 .. 10 := 0;
      Points_List_Part      : Location_List_Type;
   end record;

type Flight_Plan_Pointer_Type is access Flight_Plan_Type;

Flight_Plan         : Flight_Plan_Type;
Flight_Plan_Pointer : Flight_Plan_Pointer_Type;

Flight_Plan_Pointer                  := new Flight_Plan_Type;
Flight_Plan.Number_Of_Points_Part := Flight_Plan_Pointer.Number_Of_Points_Part;
```

INSTRUCTOR NOTES

- EXAMPLE 3
  - THIS SHOWS ALLOCATION OF AN OBJECT IN A RECORD TYPE WITH DISCRIMINANTS. IN PARTICULAR, THE DISCRIMINANTS IN THIS RECORD TYPE HAVE DEFAULT INITIAL VALUES. THIS ENSURES THAT THE DISCRIMINANTS HAVE VALUES.
  - THE NEXT TWO SLIDES DISCUSS OTHER CASES WHEN USING DISCRIMINANTS
  - IN THIS EXAMPLE, NOTE THAT Size_Part AND Priority_Part ARE GIVEN INITIAL VALUES WHEN AN OBJECT IN Buffer_Type IS ALLOCATED. ALSO, THE Length_Part IS GIVEN AN INITIAL VALUE.

ALLOCATING OBJECTS - FIRST FORM CONTINUED

- EXAMPLE 3

```
type Buffer_Type (Size_Part : Positive := 5; Priority_Part : Priority_Subtype := 1) is
   record
      Element_List_Part : Element_List_Type (1 .. Size_Part);
      Length_Part       : Natural := 0;
   end record;

type Buffer_Pointer_Type is access Buffer_Type;

Buffer_Pointer := new Buffer_Type;
```

- ALLOWED ONLY FOR RECORDS WITH DISCRIMINANTS HAVING DEFAULT INITIAL VALUES FOR THE DISCRIMINANTS

INSTRUCTOR NOTES

- THE EXAMPLES HOW A CONSTRAINT IS SPECIFIED DURING ALLOCATION.

- EXAMPLE 1
  - THIS SHOWS AN EXAMPLE OF AN ALLOCATOR WITH AN INDEX CONSTRAINT FOR AN UNCONSTRAINED ARRAY TYPE

- EXAMPLE 2
  - THIS SHOWS AN EXAMPLE OF AN ALLOCATOR WITH A DISCRIMINANT CONSTRAINT. IN THIS CASE, THERE IS NO DEFAULT VALUE FOR THE DISCRIMINANT, SO A DISCRIMINANT MUST BE SPECIFIED.

- EXAMPLE 3
  - THIS ALSO SHOWS AN EXAMPLE OF AN ALLOCATOR WITH A DISCRIMINANT CONSTRAINT. HOWEVER, IN THIS CASE THE CONSTRAINT IS OPTIONAL SINCE THE OBJECT TYPE HAS DEFAULT INITIAL VALUES FOR ITS DISCRIMINANTS (SEE PREVIOUS SLIDE).

ALLOCATING OBJECTS - SECOND FORM

new $\boxed{\text{TYPE MARK}}$ $\boxed{\text{CONSTRAINT}}$

- EXAMPLE 1

```
type String_Pointer_Type is access String;
String_Pointer : String_Pointer_Type;
   ...
String_Pointer : new String (1 .. 80);     -- ALWAYS NEEDS INDEX CONSTRAINT
```

- EXAMPLE 2

```
type Varying_String_Type (Size_Part : Natural) is
   record
      Content_Part : String (1 .. Size_Part);
      Length_Part  : Natural := 0;
   end record;

Varying_String_Pointer := new Varying_String_Type (Size_Part => 80);

-- SINCE DISCRIMINANT DOES NOT HAVE DEFAULT INITIAL VALUE,
-- ALWAYS NEED DISCRIMINANT CONSTRAINT
```

- EXAMPLE 3

```
Buffer_Pointer := new Buffer_Type (Size_Part => 20; Priority_Part => 5);
```

INSTRUCTOR NOTES

- THESE EXAMPLES SHOW ALLOCATOR WITH EXPRESSIONS.

- EXAMPLE 2

  - THIS TYPE HAS BEEN USED SEVERAL TIMES. NOTE THAT THIS TIME THE DISCRIMINANT Size_Part IS NOT BEING GIVEN A STATIC VALUE, I.E., Data_List COULD BE A PARAMETER IN SOME UNCONSTRAINED TYPE, AN OBJECT WHOSE RANGE IS NOT EVALUATED UNTIL RUNTIME, OR A FUNCTION CALL. THIS SHOULD BE CONTRASTED WITH THE SITUATION IN EXAMPLE 3.

- EXAMPLE 3

  - THE DISCRIMINANT IS USED TO SELECT A VARIANT, HENCE Reading_Is_Available MUST BE A STATIC VALUE

ALLOCATING OBJECTS - THIRD FORM

new [TYPEMARK]'([EXPRESSION])

- EXAMPLE 1

```
String_Pointer : new String'(1 .. 10 => '*');
```

- EXAMPLE 2

```
Buffer_Pointer := new Buffer_Type'
                     (Size_part         => Data_List'Length,
                      Priority_Part     => 2,
                      Element_List_Part => Data_List,
                      Length_Part       => Data_List'Length);
```

- EXAMPLE 3

```
type Sensor_Reading_Type (Valid : Boolean := False) is
   record
      case Valid is
         when False =>
            null;
         when True =>
            Reading_Part : Float := 0.0;
      end case;
   end record;

type Sensor_Reader_Pointer_Type is access Sensor_Reading_Type;

Sensor_Reading_Pointer := new Sensor_Reading_Type (Valid => Reading_Is_Available);
       --  Reading_Is_Available CANNOT BE VARIABLE OR FUNCTION CALL
```

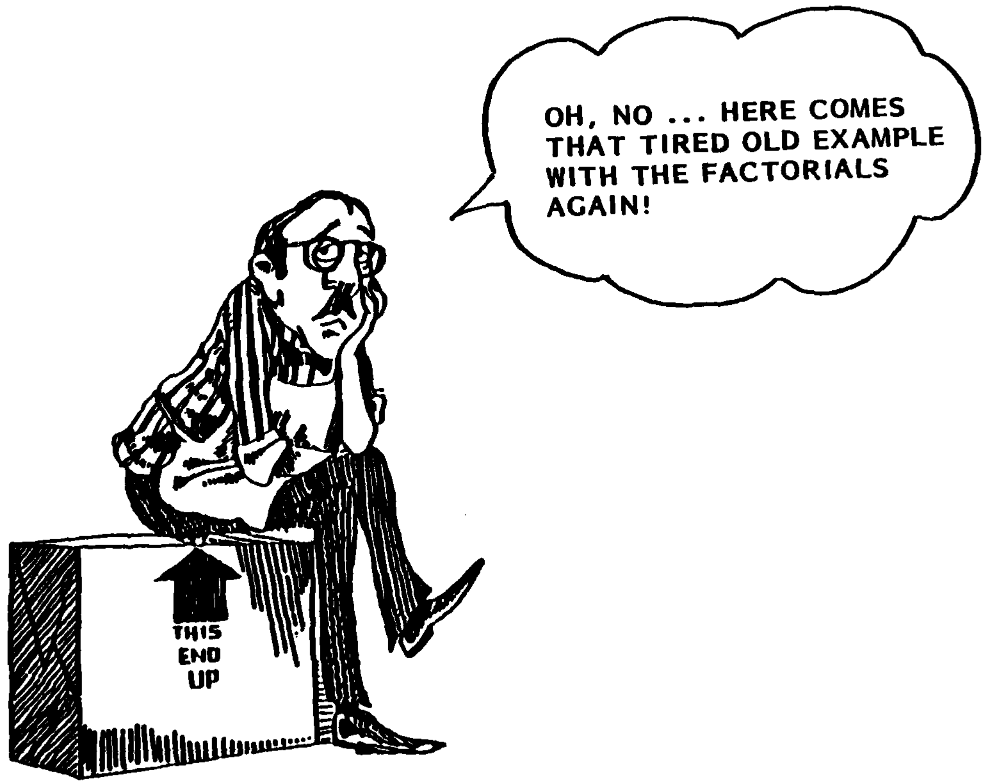
INSTRUCTOR NOTES

SECTION 2

RECURSIVE SUBPROGRAMS

INSTRUCTOR NOTES

RECURSION


OH, NO ... HERE COMES THAT TIRED OLD EXAMPLE WITH THE FACTORIALS AGAIN!
THIS END UP

INSTRUCTOR NOTES

RECURSION

RECURSION IS A METHOD IN WHICH A SUBPROGRAM SOMETIMES CALLS ITSELF AS PART OF ITS ALGORITHM. (IF TWO SUBPROGRAMS *DIRECTLY* OR *INDIRECTLY* CALL EACH OTHER, THAT IS MUTUAL RECURSION).

Ada PERMITS RECURSION. A FRESH SET OF VARIABLES IS AUTOMATICALLY PROVIDED FOR EACH NEW INVOCATION OF A SUBPROGRAM, SO THAT SEVERAL *INVOCATIONS OF* THE SAME SUBPROGRAM MAY BE IN PROGRESS SIMULTANEOUSLY WITHOUT INTERFERING WITH EACH OTHER.

INSTRUCTOR NOTES

SUCH LETTER SEQUENCES CAN MAKE A PHONE NUMBER EASIER TO REMEMBER -- "DIAL 'CALL NOW'" VERSUS "DIAL 225-5669." THIS PROGRAM PRINTS A LIST OF ALL 2,187 (= 3**7) POSSIBLE SEQUENCES, SO THAT THE USER CAN SCAN THE LIST FOR A GOOD MNEMONIC.

A PROBLEM TO BE SOLVED

1	ABC 2	DEF 3
GHI 4	JKL 5	MNO 6
PRS 7	TUV 8	WXY 9
*	OPER 0	#

GIVEN A PHONE NUMBER WITH EACH DIGIT IN THE RANGE 2 TO 9,

PRINT A LIST OF ALL POSSIBLE LETTER SEQUENCES CORRESPONDING TO THE PHONE NUMBER

EXAMPLE:

225-5669 ⇒

```
AAJJMMW
AAJJMMX
AAJJMMY
AAJJMNW
AAJJMNX
AAJJMNY
  ...
CALLNOW
  ...
CCLLONW
CCLLONX
CCLLONY
CCLLOOW
CCLLOOW
CCLLOOW
CCLLOOX
CCLLOOY
```

INSTRUCTOR NOTES

BULLET 2: THIS SPECIFICATION WILL SEEM CONTRIVED AND ARTIFICIALLY COMPLEX AT FIRST. (WHY BOTHER WITH THE Opening_Letters PARAMETER, ESPECIALLY IF IT'S ALWAYS GOING TO BE THE EMPTY STRING?)

ASK STUDENTS TO TAKE IT ON FAITH THAT THE USEFULNESS OF THE Opening_Letters PARAMETER WILL BECOME EVIDENT SHORTLY, AND IT WILL ACTUALLY SIMPLIFY THE SOLUTION.

A SOLUTION

- DEFINE THE FOLLOWING TYPES:

```
type Encodable_Digit_Type is range 2 .. 9;

type Encodable_Digit_Sequence_Type is
   array (Positive range <>) of Encodable_Digit_Type;
```

- IMPLEMENT THE FOLLOWING PROCEDURE:

```
procedure Print_Encodings
   (Opening_Letters : in String;
    Closing_Digits  : in Encodable_Digit_Sequence_Type);

-- A CALL on Print_Encodings PRINTS ALL STRINGS OF
-- (Opening_Letters'Length + Closing_Digits'Length) CHARACTERS
-- CONSISTING OF THE LETTERS IN Opening_Letters FOLLOWED
-- BY A POSSIBLE PHONE-DIAL-ENCODING OF THE
-- DIGITS IN Closing_Digits

-- EXAMPLE:  THE CALL Print_Encodings ("CALLN", (6,9)) PRINTS
--           THE FOLLOWING 9 STRINGS:
--
--            CALLNMW  CALLNMX  CALLNMY
--            CALLNNW  CALLNNX  CALLNNY
--            CALLNOW  CALLNOX  CALLNOY
```

- TO PRINT ALL THE LETTER SEQUENCES FOR DIGIT SEQUENCE Phone_Number, ISSUE THE FOLLOWING CALL:

```
Print_Encodings
   (Opening_Letters => "",
    Closing_Digits  => Phone_Number);
```

INSTRUCTOR NOTES

BULLET 2 PROVIDES THE RATIONALE FOR THE Opening_Letters PARAMETER.

RECURSION MAY BE APPROPRIATE WHENEVER A PROBLEM CAN BE DECOMPOSED INTO OTHER INSTANCES OF THE SAME PROBLEM. [LATER ON WE'LL DISCUSS ANOTHER REQUIREMENT: THAT THOSE OTHER INSTANCES BE "EASIER."]

AN ANALYSIS OF THE PROBLEM

- THE STRINGS TO BE PRINTED BY, FOR EXAMPLE, THE CALL

```
Print_Encodings ("CAL", (5, 6, 6, 9));
```

  CAN BE DIVIDED INTO THREE GROUPS:

  1. STRINGS IN WHICH THE 5 BECOMES 'J'
  2. STRINGS IN WHICH THE 5 BECOMES 'K'
  3. STRINGS IN WHICH THE 5 BECOMES 'L'

- STRINGS IN THESE GROUPS CAN BE PRINTED BY RECURSIVE CALLS ON Print_Encodings:

```
Print_Encodings ("CALJ", (6, 6, 9)); -- GROUP 1
Print_Encodings ("CALK", (6, 6, 9)); -- GROUP 2
Print_Encodings ("CALL", (6, 6, 9)); -- GROUP 3
```

- GENERAL STRATEGY:

  - MAKE RECURSIVE CALLS IN WHICH THE FIRST DIGIT IN Closing_Digits HAS BEEN REMOVED AND A LETTER CORRESPONDING TO THAT DIGIT HAS BEEN ADDED TO THE END OF Opening_Letters.

  - THREE RECURSIVE CALLS IN ALL, ONE FOR EACH LETTER CORRESPONDING TO THAT DIGIT.

INSTRUCTOR NOTES

THE "SOLUTION" ON THE PREVIOUS SLIDE WAS ACTUALLY INCOMPLETE BECAUSE IT FAILED TO ACCOUNT FOR THIS CASE.

- BULLET 2: LEAD THE STUDENTS THROUGH EACH OF THE THREE QUESTIONS, AND HAVE THE CLASS SUPPLY ANSWERS. IF THERE IS DISAGREEMENT, ENCOURAGE DISCUSSION AND GUIDE THE DISCUSSION TO THE APPROPRIATE CONCLUSION.

  - SUBBULLET 1: BY DEFINITION, IF Closing_Digits IS A NULL ARRAY, Closing_Digits'Length = 0, SO Opening_Letters'Length + Closing_Digits'Length = Opening_Letters'Length.

  - SUBBULLET 2: AN ENCODING OF A SEQUENCE OF DIGITS CONTAINS ONE LETTER FOR EACH DIGIT IN THE SEQUENCE, SO AN ENCODING OF THE EMPTY SEQUENCE OF DIGITS CONTAINS ZERO LETTERS. THE ENCODING MUST THEREFORE BE THE EMPTY STRING.

  - SUBBULLET 3: BY SUBSTITUTING THE CONCLUSIONS ABOVE INTO THE COMMENT, "A CALL ON Print_Encodings PRINTS ALL STRINGS OF LENGTH Opening_Letters'Length CONSISTING OF THE LETTERS IN Opening_Letters FOLLOWED BY THE EMPTY STRING."

  - THE ONLY SUCH STRING IS Opening_Letters ITSELF. THEREFORE, WHEN CALLED WITH A NULL ARRAY AS ITS SECOND PARAMETER, Print_Encodings SHOULD SIMPLY PRINT ITS FIRST PARAMETER.

A SPECIAL CASE

- WHAT SHOULD WE DO WHEN Closing_Digits IS A NULL ARRAY?
  - PREVIOUS ANALYSIS INAPPLICABLE BECAUSE THERE IS NO "FIRST DIGIT IN Closing_Digits".

- WHAT DOES THE SPECIFICATION OF Print_Encodings TELL US TO DO?

```
procedure Print_Encodings
   (Opening_Letters : in String;
    Closing_Digits  : in Encodable_Digit_Sequence_Type);

-- A CALL on Print_Encodings PRINTS ALL STRINGS OF
-- (Opening_Letters'Length + Closing_Digits'Length) CHARACTERS
-- CONSISTING OF THE LETTERS IN Opening_Letters FOLLOWED
-- BY A POSSIBLE PHONE-DIAL-ENCODING OF THE
-- DIGITS IN Closing_Digits
```

  - WHAT IS Opening_Letters'Length + Closing_Digits'Length?
  - WHAT ARE THE POSSIBLE PHONE DIAL ENCODINGS OF THE DIGITS IN AN EMPTY ARRAY?
  - WHAT ARE THE STRINGS THAT Print_Encodings SHOULD PRINT?

INSTRUCTOR NOTES

THIS IS JUST A STRAIGHTFORWARD IMPLEMENTATION OF THE PRECEDING ANALYSIS.

THE OTHER if STATEMENT DETECTS THE SPECIAL CASE DISCUSSED ON THE PREVIOUS SLIDE, AND PRINTS Opening_Letters WHEN Closing_Digits IS NULL.

OTHERWISE, THE FIRST DIGIT OF Closing_Digits IS PLACED IN Leading_Digit. Letter_Table (Leading_Digits) CONTAINS A STRING OF THE THREE LETTERS CORRESPONDING TO THIS DIGIT. THIS STRING IS ASSIGNED TO Eligible_Letters.

THE FIRST PARAMETER IN THE CALL ON Print_Encodings IS Opening_Letters WITH ONE OF THE LETTERS IN Eligible_Letters CATENATED ONTO THE END. THE SECOND PARAMETER, AS IMPOSING AS IT LOOKS, IS SIMPLY Closing_Digits WITH THE FIRST COMPONENT REMOVED.

Print_Encodings PROCEDURE BODY

```
with Text_IO, use Text_IO;

procedure Print_Encodings
   (Opening_Letters : in String;
    Closing_Digits  : in Encodable_Digit_Sequence_Type) is

   Letter_Table: constant array (Encodable_Digit_Type) of String (1 .. 3) :=
                    ("ABC, "DEF", "GHI", "JKL", "MNO", "PRS", "TUV", "WXY");

   Leading_Digit    : Encodable_Digit_Type;
   Eligible_Letters : String (1 .. 3);

begin -- Print_Encodings

   if Closing_Digits'Length = 0 then
      Put_Line (Opening_Letters);
   else
      Leading_Digit := Closing_Digits (Closing_Digits'First);
      Eligible_Letters := Letter_Table (Leading_Digit);
      for I in 1 .. 3 loop
         Print_Encodings
            ( Opening_Letters & Eligible_Letters (I),
              Closing_Digits (Closing_Digits'First+1 .. Closing_Digits'Last) );
      end loop;
   end if;

end Print_Encodings;
```

INSTRUCTOR NOTES

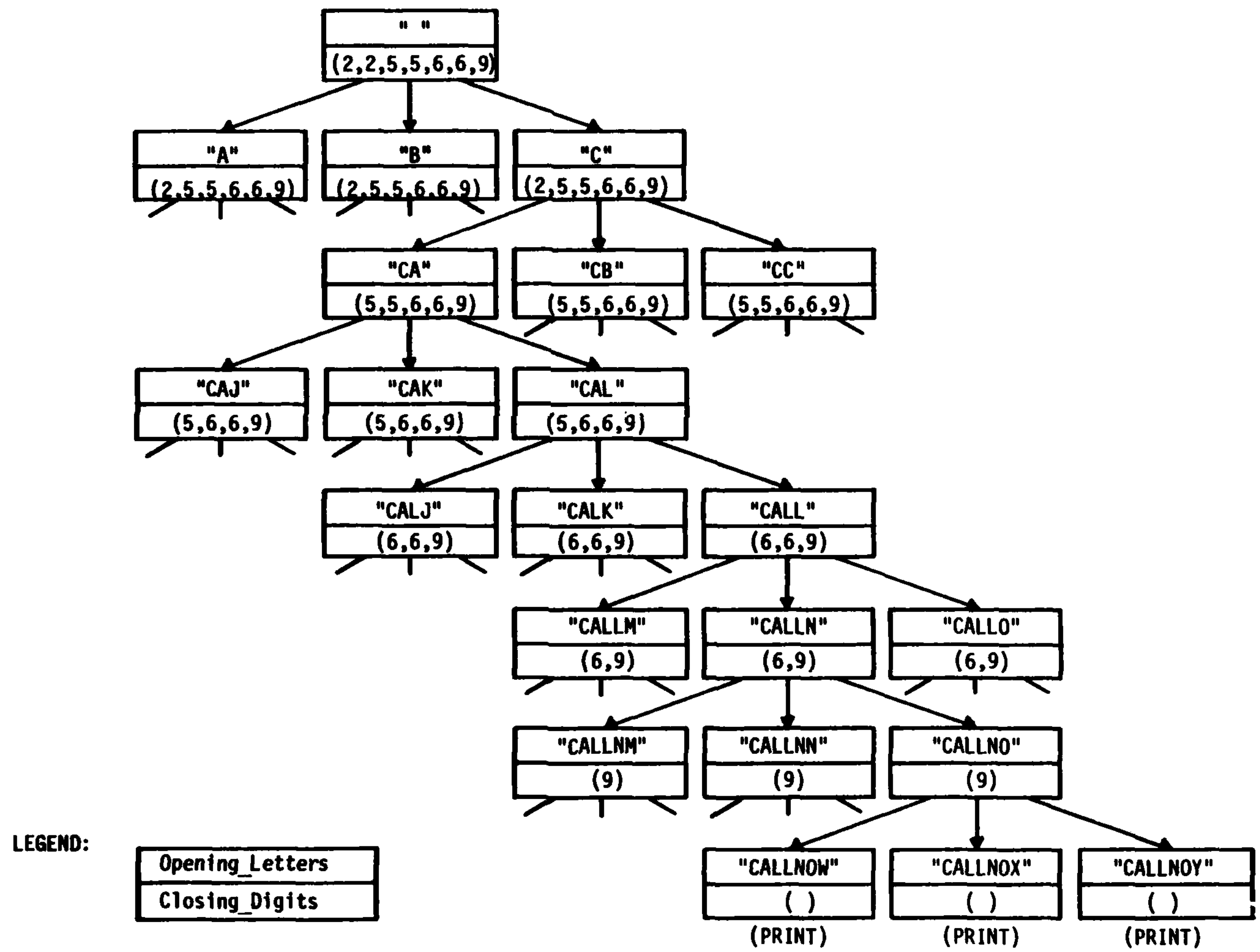
THIS DIAGRAM SHOULD HELP STUDENTS VISUALIZE HOW Print_Encodings ENUMERATES ALL POSSIBLE ENCODINGS OF A PHONE NUMBER.

EACH BOX CORRESPONDS TO A CALL ON Print_Encodings. THE BOX AT THE TOP CORRESPONDS TO THE INITIAL CALL

```
Print_Encodings (" ", (2, 2, 5, 5, 6, 6, 9));
```

THE LINES BELOW A BOX LEAD TO THE BOXES FOR THE RECURSIVE CALLS MADE BY THAT BOX'S INVOCATION OF Print_Encodings. NOT ALL BOXES ARE SHOWN, BECAUSE THERE ARE 3,280 OF THEM (INCLUDING 2,187 ON THE BOTTOM LEVEL) AND OUR SCREEN IS NOT BIG ENOUGH.

A PICTURE OF Print_Encodings IN ACTION



INSTRUCTOR NOTES

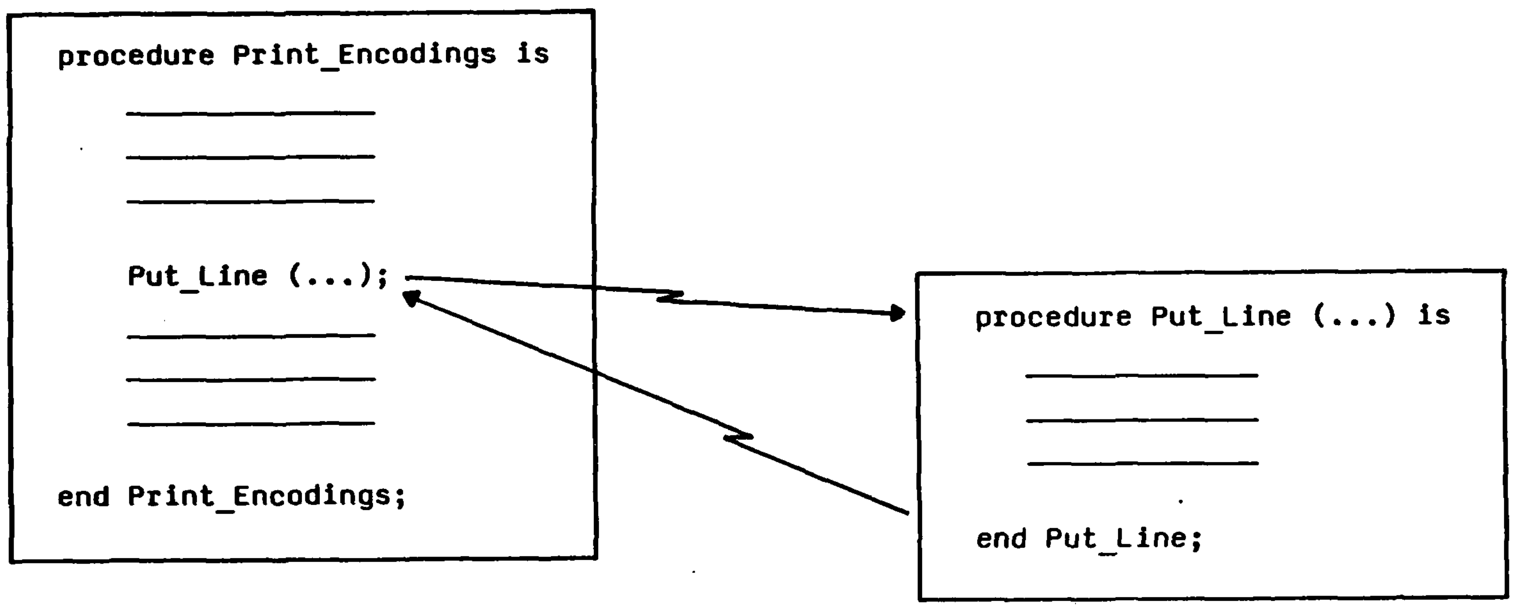
BUT HOW CAN THIS POSSIBLY WORK?

INSTRUCTOR NOTES

THE PROBLEM WITH THIS APPROACH IS THAT A RECURSIVE CALL JUMPS BACK TO THE BEGINNING OF A PROCEDURE ALREADY IN PROGRESS, AND THE VALUES THAT THE VARIABLES HELD JUST BEFORE THE RECURSIVE CALL ARE OVERWRITTEN.

SIMPLE VIEW OF PROCEDURE CALLS

WHEN OUR PROCEDURE Print_Encodings CALLS ANOTHER PROCEDURE Put_Line, THIS CAN BE VIEWED TWO WAYS. IN THE NON-RECURSIVE, "FLAT" INTERPRETATION, THE CALL AND RETURN ARE JUST JUMPS:



THIS METHOD WON'T DO FOR RECURSIVE SUBPROGRAMS.

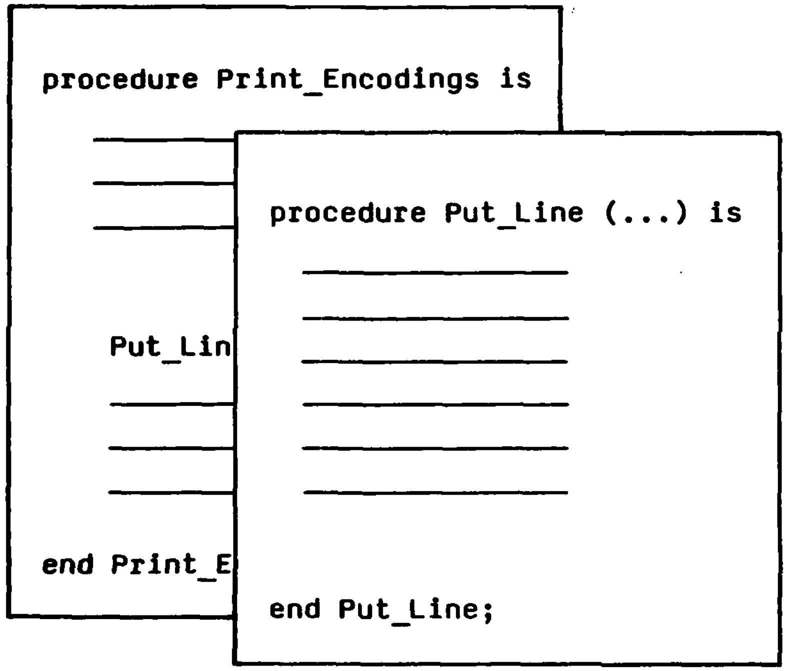
INSTRUCTOR NOTES

THE ELABORATION OF AN OBJECT DECLARATION "BRINGS OBJECTS INTO EXISTENCE." THIS MEANS MEMORY IS ALLOCATED AND THE OBJECTS CAN BE ACCESSED. WHEN THE PROGRAM UNIT CONTAINING THE DECLARATION TERMINATES, THE OBJECTS EFFECTIVELY CEASE TO EXIST.

WHEN A SUBPROGRAM CALLS ITSELF, THE OBJECTS FROM THE ORIGINAL INVOCATION ARE STILL IN EXISTENCE. ELABORATION OF THE SUBPROGRAM'S DECLARATIONS FOR THE RECURSIVE CALL CREATES A SEPARATE SET OF VARIABLES (WITH THE SAME NAMES) FOR USE BY THE NEW INVOCATION.

MORE APPROPRIATE VIEW

CONCEPTUALLY, A COPY OF Put_Line IS MADE, AND CONTROL IS TRANSFERRED.



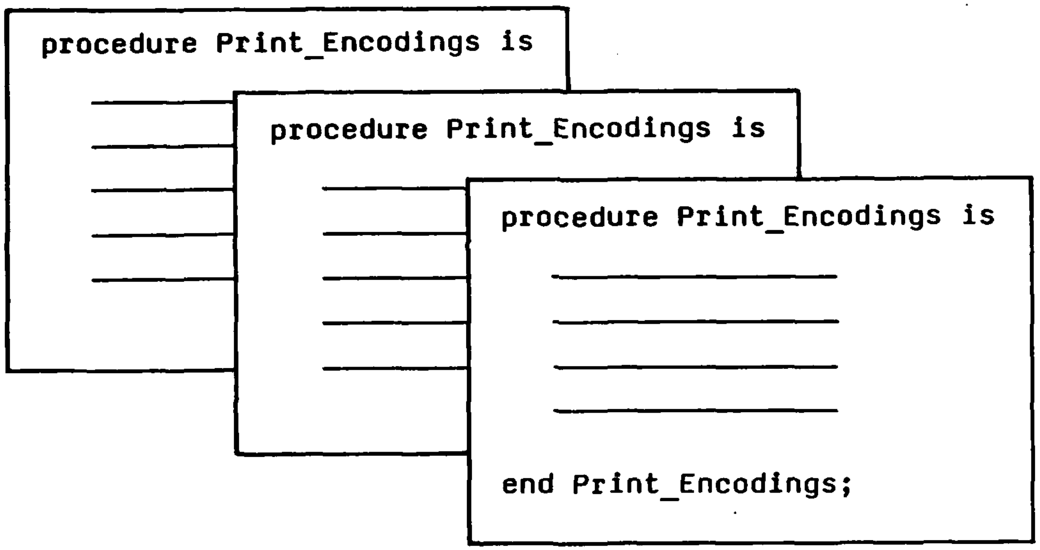
THE CALLED PROCEDURE SUPERSEDES THE CALLING PROCEDURE.

WHEN Put_Line TERMINATES, THE COPY OF Put_Line IS DISCARDED, AND Print_Encodings CONTINUES.

INSTRUCTOR NOTES

A MODEL FOR RECURSION

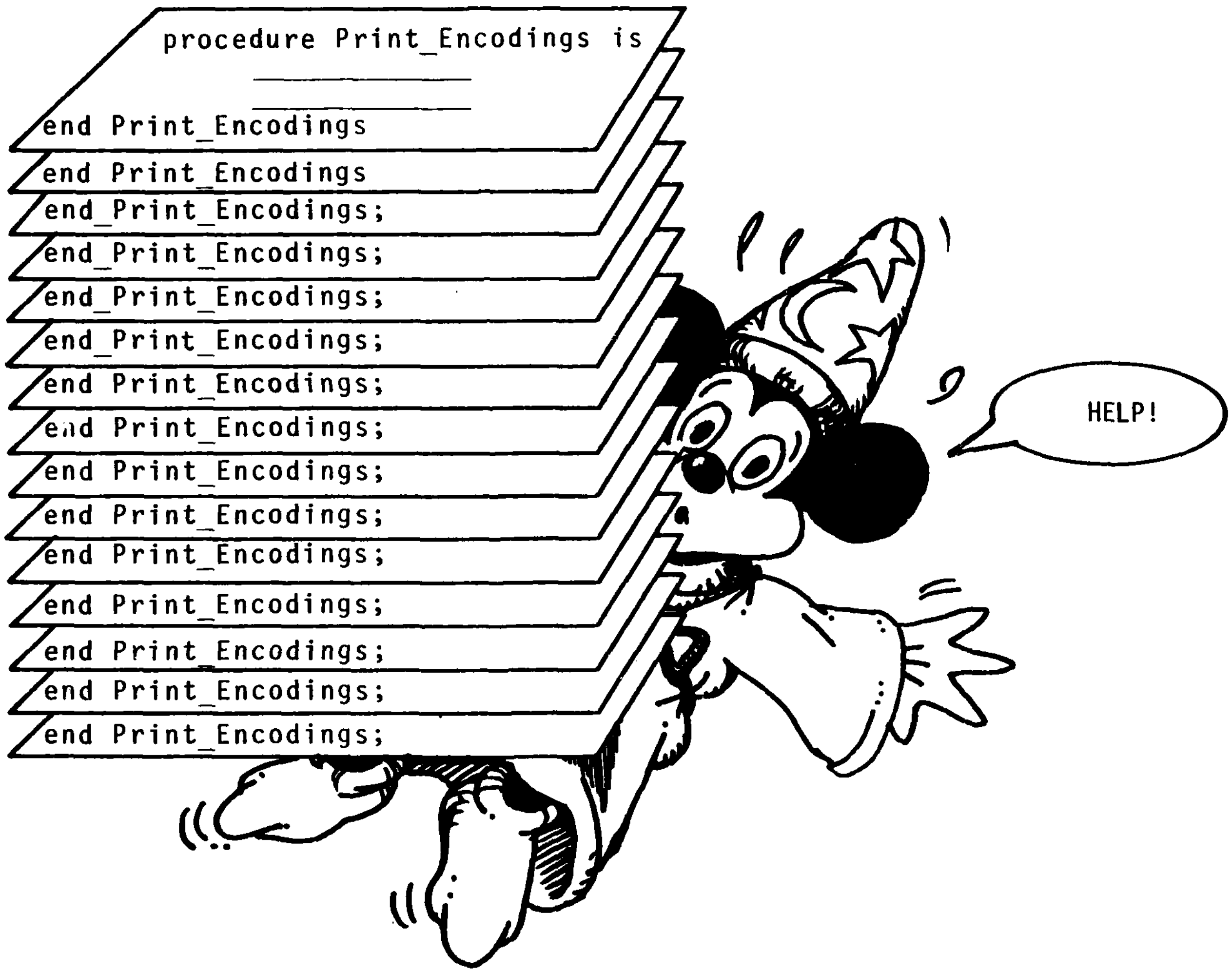
WITH THIS VIEW, THERE IS NO PROBLEM WHEN Print_Encodings CALLS ITSELF; WITH EACH CALL, A NEW COPY IS CREATED, TEMPORARILY SUPERSEDING THE EXISTING ONES:



OF COURSE, THE COMPILER NEED NOT COPY ALL OF THE CODE; A SEPARATE COPY IS ONLY NECESSARY FOR INFORMATION THAT CAN CHANGE.

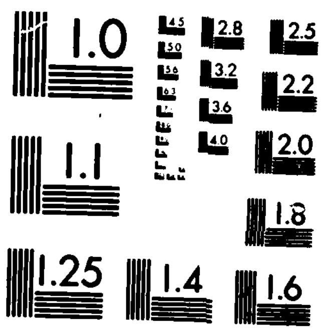
INSTRUCTOR NOTES

BUT WHY DOESN'T THIS GO ON FOREVER?



AD-A165 075   ADA (TRADEMARK) TRAINING CURRICULUM: ADVANCED ADA   2/5
TOPICS L305 TEACHER'S GUIDE VOLUME 1(U) SOFTECH INC
WALTHAM MA 1986 DAAB07-83-C-K506

UNCLASSIFIED   F/G 9/2   NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

INSTRUCTOR NOTES

BECAUSE Print_Encodings DOESN'T CALL ITSELF RECURSIVELY EVERY TIME

```
if Closing_Digits'Length = 0 then
   Put_Line (Opening_Letters);
else
   ...
end if;
```

INSTRUCTOR NOTES

TERMINATION OF RECURSIVE PROGRAMS

- THE FACT THAT Print_Encodings CAN COMPLETE WITHOUT INVOKING ANOTHER RECURSIVE CALL SHOWS THAT THE RECURSION <u>CAN</u> STOP, BUT DOESN'T PROVE THAT IT <u>WILL</u> STOP.

- IF IT IS NOT WRITTEN CORRECTLY, A RECURSIVE PROGRAM CAN KEEP CALLING ITSELF, USING MORE STORAGE FOR EACH "COPY," UNTIL THE PROGRAM RUNS OUT OF SPACE. THEN THE PROGRAM WILL RAISE THE EXCEPTION Storage_Error.

- THERE ARE WAYS WE CAN DETERMINE THAT THE RECURSION WILL EVENTUALLY STOP.

INSTRUCTOR NOTES

RECURSIVE CALLS SHOULD SOLVE "EASIER" VERSIONS OF THE SAME PROBLEM

A RECURSIVE PROGRAM WILL TERMINATE EVENTUALLY IF

- EACH RECURSIVE CALL HAS PARAMETERS DESCRIBING AN "EASIER" CASE OF THE PROBLEM TO BE SOLVED
- "SUFFICIENTLY EASY" CASES ARE SOLVED WITHOUT FURTHER RECURSION
- A PROBLEM CAN BE MADE "EASIER" ONLY A FINITE NUMBER OF TIMES BEFORE IT IS "SUFFICIENTLY EASY" TO BE SOLVED WITHOUT RECURSION.

ONE CALL ON Print_Encodings IS "EASIER" THAN ANOTHER IF IT IS CALLED WITH A SHORTER Closing_Digits ARRAY.

- EACH RECURSIVE CALL IS FOR AN "EASIER" ARRAY, OF LENGTH Closing_Digits'Length -1.
- A PROBLEM IS "SUFFICIENTLY EASY" TO BE SOLVED WITHOUT RECURSION WHEN Closing_Digits'Length = 0.
- Closing_Digits'Length CAN BE DECREASED ONLY FINITELY MANY TIMES BEFORE IT IS ZERO.

INSTRUCTOR NOTES

CONVINCING ONE'S SELF THAT A RECURSIVE SUBPROGRAM DOES WHAT IT IS SUPPOSED TO

1. WALK THROUGH THE PATHS THAT DON'T INVOLVE RECURSIVE CALLS AND CONVINCE YOURSELF THAT THOSE PATHS WORK CORRECTLY.
   (WHEN Closing_Digits IS EMPTY, THE PROPER ACTION IS TO PRINT Opening_Letters BY ITSELF.)

2. ASSUME THAT THE SUBPROGRAM WORKS CORRECTLY ON EASIER CASES, THAT IS, THAT THE RECURSIVE CALLS DO WHAT THEY ARE SUPPOSED TO.

3. CONVINCE YOURSELF THAT, GIVEN THIS ASSUMPTION, THE INVOCATION MAKING THE RECURSIVE CALLS DOES WHAT IT'S SUPPOSED TO.
   (THE STRINGS TO BE PRINTED CAN BE DIVIDED INTO THREE GROUPS. BY THE ASSUMPTION IN STEP 2 ABOVE, EACH RECURSIVE CALL CORRECTLY PRINTS THE STRINGS IN ONE OF THESE GROUPS.)

INSTRUCTOR NOTES

ADDING Decompose_Problem DOES NOT REALLY MAKE THIS PROGRAM CLEARER. IT JUST PROVIDES AN EXAMPLE OF MUTUAL RECURSION THAT WILL BE FAMILIAR BY NOW.

(THIS PROGRAM IS REALLY ILLEGAL, AS THE NEXT SLIDE WILL EXPLAIN.)

```
procedure Decompose_Problem
   (Old_Opening_Letters : in String;
    New_Letters         : in String;
    Remaining_Digits    : in Encodable_Digit_Sequence_Type) is
begin
   for I in 1 .. 3 loop
      Print_Encodings (Old_Opening_Letters & New_Letters (I), Remaining_Digits);
   end loop;
end Decompose_Problem;

procedure Print_Encodings
   (Opening_Letters : in String;
    Closing_Digits  : in Encodable_Digit_Sequence_Type) is

   Letter_Table : constant array (Encodable_Digit_Type) of String :=
                  ("ABC", "DEF", "GHJ", "JKL", "MNO", "PRS", "TUV", "WXY");

   Leading_Digit    : Encodable_Digit_Type;
   Eligible_Letters : String (1 .. 3);

begin
   if Closing_Digits'Length = 0 then
      Put_Line (Opening_Letters);
   else
      Leading_Digit    := Closing_Digits (Closing_Digits'First);
      Eligible_Letters := Letter_Table (Leading_Digit);
      Decompose_Problem
         (Old_Opening_Letters => Opening_Letters,
          New_Letters         => Eligible_Letters
          Remaining_Digits    => Closing_Digits
                                 (Closing_Digits'First+1 .. Closing_Digits'Lost));
   end if;
end Print_Encodings;
```

- Print_Encodings WORKS JUST AS BEFORE, BUT THE for LOOP CONTAINING THE RECURSIVE CALL HAS BEEN BROKEN OFF INTO ANOTHER SUBPROGRAM, Decompose_Problem.
- NOW Print_Encodings CALLS Decompose_Problem AND Decompose_Problem CALLS Print_Encodings.

INSTRUCTOR NOTES

- BULLET 2: REVERSING THE TWO BODIES WOULD MAKE THE CALL ON Decompose_Problem INSIDE THE Print_Encodings BODY ILLEGAL.

MUTUAL RECURSION REQUIRES SUBPROGRAM DECLARATIONS

- Ada RULE: A SUBPROGRAM MAY NOT BE CALLED UNTIL EITHER A DECLARATION OR BODY FOR THAT SUBPROGRAM HAS BEEN ENCOUNTERED:

```
procedure Decompose_Problem (...) is
   ...
begin
   ...
   Print_Encodings (...); -- ILLEGAL CALL
   ...
end Decompose_Problem;

procedure Print_Encodings (...) is -- SCOPE OF Print_Encodings STARTS HERE
   ...
begin
   ...
      Decompose_Problem (...);
end Print_Encodings;
```

- WHAT WOULD HAPPEN IF WE REVERSED THE ORDER OF THE TWO PROCEDURE BODIES?

INSTRUCTOR NOTES

- REVIEW SUBPROGRAM DECLARATIONS. THE TEXT BEGINNING "function" OR "procedure" FOLLOWED BY THE SUBPROGRAM NAME, FORMAL PARAMETER LIST (IF ANY), AND RESULT TYPE FOR FUNCTIONS IS CALLED A <u>SUBPROGRAM SPECIFICATION</u>. A SUBPROGRAM DECLARATION IS A SUBPROGRAM SPECIFICATION FOLLOWED BY A SEMICOLON. A SUBPROGRAM BODY BEGINS WITH A SUBPROGRAM SPECIFICATION FOLLOWED BY THE WORD IS.

  WE HAVE ALREADY SEEN SUBPROGRAM DECLARATIONS IN PACKAGE SPECIFICATIONS, TO DECLARE SUBPROGRAMS THAT THE PACKAGE IS PROVIDING TO THE OUTSIDE WORLD. THIS IS ANOTHER USE FOR THEM.

  THE SCOPE OF A SUBPROGRAM STARTS WITH THE SUBPROGRAM DECLARATION IF THERE IS ONE, WITH THE SUBPROGRAM BODY OTHERWISE.

MUTUAL RECURSION REQUIRES SUBPROGRAM DECLARATIONS (Continued)

- SOLUTION:

```
  procedure Print_Encodings                                     -- PROCEDURE DECLARATION
     (Opening_Letters : in String;
      Closing_Digits  : in Encodable_Digit_Sequence_Type);      -- SCOPE OF Print_Encodings
                                                                -- STARTS HERE
  procedure Decompose_Problem (...) is
     ...
  begin
     Print_Encodings (...);                                     -- NOW LEGAL
     ...
end Decompose_Problem;

procedure Print_Encodings                                       -- PROCEDURE BODY
   (Opening_Letters : in String;
    Closing_Digits  : in Encodable_Digit_Sequence_Type) is      -- (REPEAT SAME INFORMATION)
   ...
begin
   ...
     Decompose_Problem (...);
   ...
end Print_Encodings;
```

INSTRUCTOR NOTES

PART II

FUNDAMENTAL DATA STRUCTURES

INSTRUCTOR NOTES

SECTION 3

SETS USING BOOLEAN ARRAYS

INSTRUCTOR NOTES

MANY PROBLEMS ARE EASILY CHARACTERIZED IN TERMS OF SETS OF ITEMS BELONGING TO SOME FINITE UNIVERSE. THE FOLLOWING PROBLEM IS AN EXAMPLE

THERE ARE TWENTY-THREE MESSAGE RELAY STATIONS LOCATED ACROSS THE MAINLAND U.S. THESE ARE REPRESENTED BY VALUES IN THE ENUMERATION TYPE Station_Type.

THERE IS A FUNCTION Working_Connection THAT TAKES TWO Station_Type VALUES AND RETURNS A BOOLEAN VALUE INDICATING WHETHER THE FIRST STATION CAN TRANSMIT DIRECTLY TO THE SECOND.

LET US ASSUME THAT THE TYPE Station_Type AND THE FUNCTION Working_Connection ARE DEFINED IN THE SEPARATELY COMPILED PACKAGE Station_Package.

SIX OF THE RELAY STATIONS CAN ONLY HANDLE LOW-SPEED TRANSMISSIONS.

SIX OF THE STATIONS ARE NOT PERMITTED TO HANDLE HIGH-SECURITY MESSAGES.

THE PROBLEM IS TO FIND ALL STATIONS THAT CAN, DIRECTLY OR INDIRECTLY, RECEIVE A HIGH SPEED, HIGH_SECURITY TRANSMISSION ORIGINATING AT THE SARASOTA STATION.

THIS PROBLEM CAN BE SOLVED IN TERMS OF SETS OF MESSAGE RELAY STATIONS. THIS IS NOT NECESSARILY THE MOST EFFICIENT SOLUTION, BUT IT ILLUSTRATES HOW OPERATIONS ON SUCH SETS CAN BE IMPLEMENTED IN ADA.

PROBLEM

- A SET OF MESSAGES RELAY STATIONS

```
type Station_Type is
   (Atlanta_Station, Bangor_Station, Bismarck_Station, Boise_Station,
   Cedar_Rapids_Station, Corpus_Christi_Station, Duluth_Station,
   Fort_Worth_Station, Fresno_Station, Great_Falls_Station, Jackson_Station,
   Lansing_Station, Los_Angeles_Station, Louisville_Station, Omaha_Station,
   Phoenix_Station, Richmond_Station, Santa_Fe_Station, Sarasota_Station,
   Seattle_Station, St_Louis_Station, Syracuse_Station, Wichita_Station);
```

- A FUNCTION INDICATING WHETHER IT IS POSSIBLE FOR ONE STATION TO TRANSMIT DIRECTLY TO THE SECOND

```
      function Working_Connection (From, To: Station_Type) return Boolean;
```

- A DESIGNATED SET OF STATIONS THAT CAN ONLY HANDLE LOW-SPEED TRANSMISSIONS.

- A DESIGNATED SET OF STATIONS THAT CANNOT HANDLE CLASSIFIED TRANSMISSIONS.

<u>WHICH STATIONS CAN BE CONTACTED FROM THE SARASOTA STATION, ASSUMING THAT MESSAGES ARE ONLY SENT THROUGH HIGH-SPEED, HIGH-SECURITY STATIONS?</u>

INSTRUCTOR NOTES

THIS PROGRAM IS WRITTEN IN A COMBINATION OF ADA AND MATHEMATICAL NOTATION. LATER SLIDES WILL EXPLAIN HOW TO EXPRESS THE SET CONCEPTS IN ADA. THE SET NOTATION IS SURROUNDED BY BOXES.

THE DECLARATION OF Station_IO SHOULD BE EXPLAINED AT THIS POINT AS STEP 2 OF THE I/O PROCESS PRESENTED IN L202 TO ALLOW Station_Type VALUES TO BE PRINTED (AS CAPITALIZED ENUMERATION LITERALS).

THE DEFINITION OF Station_Set_Type WILL BE GIVEN LATER. FOR NOW, WE ARE CONCERNED WITH HOW SETS OF RELAY STATIONS WILL BE USED TO SOLVE THE PROBLEM, NOT WITH HOW WE REPRESENT THE SETS.

Low_Speed_Stations AND Low_Security_Stations ARE CONSTANT SETS. THE CURLY BRACES ARE NOT PART OF ADA. THEY DESIGNATE THE OPERATION OF SPECIFYING A SET BY LISTING ITS ELEMENTS. Root_Station IS A CONSTANT INDICATING THE STATION FROM WHICH THE TRANSMISSION IS ASSUMED TO ORIGINATE. High_Security_Stations, High_Speed_Stations, ETC. ARE VARIABLES WHOSE VALUES WILL BE SETS OF STATIONS. IN CONTRAST, S IS A VARIABLE WHOSE VALUES WILL BE INDIVIDUAL STATIONS.

SOLUTION USING SETS

```
with Text_IO, Station_Package; use Station_Package;

procedure Find_Recipients is

   -- Station_Package defines Station_Type and Working_Connection
   package Station_IO is new Text_IO.Enumeration_IO (Station_Type);
   type Station_Set_Type is ???;
   Low_Speed_Stations : constant Station_Set_Type :=
                      |Atlanta_Station, Boise_Station,
                       Great_Falls_Station, Los_Angeles_Station,
                       Omaha_Station, Santa_Fe_Station|  ;

   Low_Security_Stations : constant Station_Set_Type :=
                      |Atlanta_Station, Fort_Worth_Station,
                       Fresno_Station, Los_Angeles_Station,
                       Richmond_Station, St_Louis_Station|  ;

   Root_Station : constant Station_Type := Sarasota_Station;
   High_Security_Stations,
      High_Speed_Stations,
      Eligible_Stations,
      Stations_To_Be_Processed,
      Recipients_Found,
      Possible_New_Recipients,
      New_Recipients              : Station_Set_Type;
   S                              : Station_Type;

begin -- Find_Recipients

      statements on next slide

end Find_Recipients;
```

THE BARS IN THE FIRST TWO ASSIGNMENT STATEMENTS DESIGNATE SET COMPLEMENT OPERATION. High_Security_Stations IS ASSIGNED THE SET CONSISTING OF ALL Station_Type VALUES THAT ARE NOT MEMBERS OF THE SET Low_Security_Stations. SIMILARLY, High_Security_Stations IS ASSIGNED THE SET CONSISTING OF ALL Station_Type VALUES THAT ARE NOT MEMBERS OF THE SET Low_Speed_Stations.

THE ONLY STATIONS RELEVANT TO SOLVING THE PROBLEM ARE THOSE THAT ARE MEMBERS OF BOTH High_Security_Stations AND High_Speed_Stations. THE INTERSECTION OF THE SETS High_Security_Stations AND High_Speed_Stations IS A SET CONSISTING OF PRECISELY THOSE COMMON ELEMENTS. THE THIRD ASSIGNMENT STATEMENT ASSIGNS THIS INTERSECTION TO THE SET VARIABLE Eligible_Stations.

THE NEXT TWO ASSIGNMENT STATEMENTS INITIALIZE Stations_To_Be_Processed AND Recipients_Found FOR USE INSIDE THE FOLLOWING LOOP. AT THE BEGINNING OF EACH PASSAGE THROUGH THE LOOP, Recipients_Found WILL HOLD THE SET OF STATIONS THAT HAVE BEEN ESTABLISHED SO FAR AS RECIPIENTS OF THE MESSAGE (THE ORIGINATING STATION IS AUTOMATICALLY CONSIDERED A RECIPIENT). WHEN THE LOOP TERMINATES, THE VARIABLE WILL HOLD THE "ANSWER" TO THE PROBLEM. SIMILARLY, AT THE BEGINNING OF EACH PASSAGE THROUGH THE LOOP, Stations_To_Be_Processed WILL BE THE SET OF STATIONS THAT HAVE BEEN ESTABLISHED AS RECIPIENTS OF THE MESSAGE, BUT HAVE NOT BEEN EXAMINED YET TO SEE WHOM THEY CAN PASS THE MESSAGE ON TO. THE TEST FOR EXITING THE LOOP IS MADE AT THE BOTTOM OF THE LOOP.

SOLUTION USING SETS (CONTINUED)

```
with Text_IO, Station_Package; use Station_Package;
procedure Find_Recipients is
     declarations on previous slide
begin -- Find_Recipients
   High_Security_Stations := Low_Security_Stations ;        -- complement
   High_Speed_Stations    := Low_Speed_Stations ;
   Eligible_Stations :=
     High_Security_Stations ∩ High_Speed_Stations ;          -- intersection
   Stations_To_Be Processed := {Root_Station} ;              -- list of elements
   Recipients_Found := {Root_Station} ;
```

THE FIRST ASSIGNMENT STATEMENT ASSIGNS THE EMPTY SET -- THE SET CONTAINING NO ELEMENTS -- TO THE VARIABLE New_Recipients. THIS IS A SPECIAL CASE OF THE OPERATION SPECIFYING A SET BY LISTING ITS ELEMENTS. EACH NEW MESSAGE RECIPIENT DISCOVERED DURING THIS PASSAGE THROUGH THE LOOP WILL BE ADDED TO THE SET New_Recipients AS IT IS DISCOVERED.

THE EXTRACTION OPERATION SELECTS AN ARBITRARY VALUE FROM Stations_To_Be_Processed, REMOVES THAT STATION FROM THE SET, AND PLACES THAT VALUE IN THE Station_Type VARIABLES.

THE PROGRAM WILL NOW PROCESS SET S TO SEE WHICH NEW STATIONS CAN BE REACHED FROM RECIPIENT S. WE ARE NOT INTERESTED IN STATIONS THAT CAN BE REACHED FROM S BUT HAVE ALREADY BEEN SHOWN TO BE RECIPIENTS BY SOME OTHER ROUTE. THUS Possible_New_Recipients IS ASSIGNED THE SET OF ALL HIGH-SPEED, HIGH-SECURITY STATIONS THAT HAVE NOT BEEN ESTABLISHED AS RECIPIENTS SO FAR. THE MINUS SIGN DENOTES SET DIFFERENCE: Eligible_Stations-Recipients_Found IS THE SET OF ALL STATIONS THAT ARE MEMBERS OF Eligible_Stations BUT NOT OF Recipients_Found.

THE ACTION OF PERFORMING SOME OPERATION FOR EACH MEMBER OF A SET IS DENOTED BY A SPECIAL VARIETY OF FOR-LOOP. AGAIN, THIS IS A SPECIAL SET NOTATION, NOT PART OF ADA. A LATER SLIDE WILL SHOW HOW TO IMPLEMENT THIS OPERATION IN ADA. X ACTS JUST LIKE THE LOOP PARAMETER IN A GENUINE FOR-LOOP. THE LOOP IS EXECUTED WITH X SET TO EACH ELEMENT OF Possible_New_Recipients.

THE PURPOSE OF THE LOOP IS TO CHECK EACH STATION IN Possible_New_Recipients TO SEE IF IT CAN BE REACHED FROM S, AND TO ADD THAT STATION TO New_Recipients IF IT CAN BE. THE SYMBOL U REPRESENTS THE UNION OPERATION, WHICH COMPUTES THE SET OF STATIONS THAT BELONG TO EITHER OR BOTH OF ITS OPERANDS. THE ASSIGNMENT STATIONS IN THE LOOP CONTAINS A SPECIAL CASE OF THE UNION OPERATION, WHICH HAS THE EFFECT OF ADDING A SINGLE STATION TO A SET. WE CALL THIS INSERTION.

AFTER THE LOOP, TWO MORE UNION OPERATIONS ADD ALL NEWLY DISCOVERED RECIPIENTS TO THE SET OF ALL RECIPIENTS FOUND SO FAR AND THE SET OF STATIONS REQUIRING FURTHER PROCESSING. THE LOOP TERMINATES AFTER A PASSAGE IN WHICH THE VALUE EXTRACTED FROM S IS THE LAST MEMBER OF S, AND NO NEW STATIONS ARE REACHABLE FROM S (SO New_Recipients = { }).

FINALLY, ONCE Recipients_Found CONTAINS ALL THE STATIONS IT SHOULD, A PSEUDO-FOR_LOOP PRINTS EACH MEMBER OF THE SET.

SOLUTION USING SETS (CONTINUED)

```
loop
   New_Recipients := { };                                    -- empty set
   extract a value S from Stations_To_Be_Processed ;         -- extraction
   Possible_New_Recipients :=
      Eligible_Stations - Recipients_Found  ;                -- set difference
   for each element X in Possible_New_Recipients loop        -- for each element
      if Working_Connection (From => S, To => X) then
         New_Recipients := New_Recipients U{X};              -- insertion
      end if;
   end loop;
   Recipients_Found := Recipients_Found U New_Recipients ;   -- union
   Stations_To_Be_Processed :=
      Stations_To_Be_Processed U New_Recipients ;
   exit when Stations_To_Be_Processed ={ };  -->>> LOOP EXIT <<<--
end loop;
for each element X in Recipients_Found loop
   Station_IO.Put(X);
   Text_IO.New_Line;
end loop;

end Find_Recipients;
```

BECAUSE Station_Type IS AN ENUMERATION TYPE, IT CAN BE USED AS THE INDEX TYPE OF AN ARRAY.

A SET WHOSE MEMBERS BELONG TO AN ENUMERATION TYPE CAN BE REPRESENTED AS AN ARRAY OF BOOLEAN VALUES INDEXED BY ENUMERATION TYPE VALUES. EACH BOOLEAN VALUE IN THE ARRAY INDICATES WHETHER OR NOT THE CORRESPONDING INDEX VALUE IS AN ELEMENT OF THE SET.

ASSUME STATION IS A VARIABLE OF TYPE Station_Type and Station_Set_1 IS A VARIABLE OF TYPE Station_Set_Type. THEN THE BOOLEAN EXPRESSION Station_Set_1 (Station) IS TRUE IF STATION IS A MEMBER OF Station_Set_1 AND FALSE IF IT IS NOT. THE ADA EXPRESSIONS ON THE RIGHT ARE A TRANSLATION OF THE SET OPERATIONS ON THE LEFT. (THE SET MEMBERSHIP TESTS $\in$ AND $\notin$ WERE NOT ACTUALLY USED IN THE Find_Recipients PROGRAM).

A SET SPECIFIED BY A LIST OF Station_Type LITERALS IN BRACES CAN BE TRANSLATED AS AN ARRAY AGGREGATE IN WHICH THE ELEMENTS INDEXED BY THE LISTED VALUES ARE SET TO TRUE AND ALL OTHER ELEMENTS ARE SET TO FALSE. AS A SPECIAL CASE, THE EMPTY SET CAN BE TRANSLATED AS AN AGGREGATE IN WHICH ALL ELEMENTS ARE SET TO FALSE.

IMPLEMENTATION OF SETS USING ARRAYS OF BOOLEANS

```
type Station_Set_Type is array (Station_Type) of Boolean;

Station                          : Station_Type;
Station_Set_1, Station_Set_2 : Station_Set_Type;
```

Set Operation	Implementation in Ada
Station $\in$ Station_Set_1	Station_Set_1 (Station)
Station $\not\in$ Station_Set_1	not Station_Set_1 (Station)
{Atlanta_Station, Boise_Station}	(Atlanta_Station | Boise_Station => True, others => False)
{ }	(Station_Type => False)

INSTRUCTOR NOTES

THE ADA LOGICAL OPERATIONS -- AND, OR, XOR, AND NOT -- CAN BE APPLIED TO OPERANDS IN A ONE-DIMENSIONAL ARRAY TYPE WITH BOOLEAN COMPONENTS. IN THE CASE OF AND, OR, AND XOR, THE TWO OPERANDS MUST BE OF THE SAME LENGTH, AND THE RESULT IS AN ARRAY OF THE SAME TYPE AND LENGTH. THE FIRST COMPONENT OF THE RESULT IS DETERMINED BY APPLYING THE LOGICAL OPERATION TO THE FIRST COMPONENT OF EACH OPERAND, THE SECOND COMPONENT OF EACH OPERAND, AND SO FORTH.

(RUN THROUGH INDIVIDUAL EXAMPLES ON THE SLIDE IF NECESSARY.)

THE COMPLEMENT, INTERSECTION, UNION, AND SET DIFFERENCE OPERATIONS CAN BE IMPLEMENTED USING LOGICAL OPERATIONS, SINCE Station_Set_Type IS A ONE-DIMENSIONAL ARRAY TYPE WITH BOOLEAN COMPONENTS. THE UNION AND DIFFERENCE ARE INTENDED AS AN IN-CLASS EXERCISE.

- A STATION IS A Low_Security_Station IF AND ONLY IF IT'S NOT A High_Security_Station, SO THE NOT OPERATOR COMPUTES SET COMPLEMENTS.

- A STATION IS IN High_Security_Stations ∩ High_Speed_Stations ONLY IF IT IS IN BOTH THOSE SETS, SO THE AND OPERATOR COMPUTES INTERSECTIONS.

- A STATION IS IN Recipients_Found U New_Recipients IF AND ONLY IF IT IS IN EITHER OR BOTH OF THOSE SETS, SO THE OR OPERATOR COMPUTES UNIONS.

  ANSWER: Recipients_Found or New_Recipients

- A STATION IS IN Eligible_Recipients - Recipients_Found ONLY IF IT IS A MEMBER OF Eligible_Recipients BUT NOT OF Recipient_Found. THUS Eligible Stations AND NOT Recipient_Found COMPUTES THIS SET DIFFERENCE.

  ANSWER: Eligible_Stations and not Recipients_Found

IMPLEMENTATION OF SETS USING ARRAYS OF BOOLEANS (CONTINUED)

Componentwise Application of Logical Operators to Boolean Arrays:

```
(True, True, False, False) and (True, False, True, False) = (True, False, False, False)
(True, True, False, False) or  (True, False, True, False) = (True, True, True, False)
(True, True, False, False) xor (True, False, True, False) = (False, True, True, False)
not (True, False)                                         = (False, True)
```

Set Operation	Implementation in Ada
$\overline{\text{Low\_Security\_Stations}}$	not Low_Security_Stations
High_Security_Stations ∩ High_Speed_Stations	High_Security_Stations and High_Speed_Stations
Recipients_Found ∪ New_Recipients	
Eligible_Stations - Recipients_Found	

INSTRUCTOR NOTES

THE OPERATOR INSERTING A SINGLE ELEMENT INTO A SET IS MOST SIMPLY TRANSLATED INTO AN ASSIGNMENT SETTING THE CORRESPONDING ARRAY ELEMENT TO TRUE.

THE OPERATOR EXTRACTING AN ARBITRARY ELEMENT FROM A SET IS ONLY WELL-DEFINED WHEN THE SET IS NOT EMPTY. IN THIS CASE, THE ARRAY HAS AT LEAST ONE COMPONENT EQUAL TO TRUE. THE WHILE LOOP ON THE SLIDE SEARCHES FOR THE FIRST SUCH COMPONENT, SETTING S TO ITS INDEX. (THE INDEX IS A VALUE IN Station_Type, REPRESENTING THE VALUE TO BE EXTRACTED.) THE COMPONENT ITSELF IS SET TO FALSE TO REMOVE IT FROM THE SET.

THE PSEUDO-FOR-LOOP EXECUTED FOR EACH ELEMENT OF A SET IS TRANSLATED AS A FOR-LOOP EXECUTED FOR EACH ELEMENT OF THE UNIVERSE, I.E., EACH VALUE IN Station_Type, AND CONTAINING AN IF-STATEMENT TO TEST WHETHER EACH VALUE OF THE LOOP PARAMETER IS A MEMBER OF THE SET BEFORE EXECUTING THE BODY OF THE PSEUDO-LOOP. (THIS TRANSLATION PRESUMES THAT THE SET CONTROLLING THE LOOP IS NOT MODIFIED BY THE BODY OF THE LOOP.)

IMPLEMENTATION OF SETS USING ARRAYS OF BOOLEANS (CONTINUED)

Set Operation	Implementation in Ada
`New_Recipients U {X} ;`	`New_Recipients (X) := True;`

`extract a value S from Stations_To_Be_Processed;`
`(assuming Stations_To_Be_Processed /= {})`
	`S := Station_Type'First;`
	`while not Stations_To_Be_Processed(S) loop`
	`   S := Station_Type'Succ (S);`
	`end loop;`
	`Stations_To_Be_Processed (S) := False;`

`for each element X in Recipients_Found loop`
`   ...`
`end loop;`
	`for X in Station_Type loop`
	`   if Recipients_Found (X) then`
	`      ...`
	`   end if;`
	`end loop;`

INSTRUCTOR NOTES

SECTION 4

LINEAR LISTS

INSTRUCTOR NOTES

A MORE FAMILIAR USE OF ARRAYS IS AS A <u>LIST</u> OF ITEMS. THE ITEMS LISTED ARE STORED IN THE COMPONENTS OF THE ARRAY.

IF THE SIZE OF THE LIST IS FIXED, THE REPRESENTATION OF THE LIST IS SIMPLE. HERE List_Type IS DECLARED AS A TYPE. VALUES IN THE TYPE ARE LISTS OF EXACTLY TEN ITEMS OF SOME TYPE CALLED Item_Type.

AMONG THE OPERATIONS ON FIXED-LENGTH LISTS ARE EXAMINING OR MODIFYING THE ITEM IN A PARTICULAR POSITION OR PERFORMING SOME OPERATION FOR EACH ELEMENT OF THE LIST IN TURN.

THE 'Range ATTRIBUTE PROVIDES A CONVENIENT ELEGANT WAY OF SPECIFYING THAT A LOOP IS TO BE EXECUTED FOR EACH VALID INDEX VALUE IN TURN. IN THIS CASE, WRITING THE ATTRIBUTE L'Range IS EQUIVALENT TO WRITING 1 .. 10.

ONE-DIMENSIONAL ARRAYS AS FIXED LENGTH LISTS

CONTEXT

```
-- P HAS A VALUE BETWEEN 1 AND 10.
type Item_Type is range 1 .. 100;
type List_Type is array (1 .. 10) of Item_Type;
List : List_Type;
Item, Value : Item_Type;
```

TYPICAL OPERATIONS

- USE VALUE STORED AT POSITION P:

```
Item := List (P) + 1;
```

- REPLACE VALUE STORED AT POSITION P:

```
List (P) := Value;
```

- PERFORM SOME OPERATION FOR EACH ITEM IN THE LIST:

```
for I in List'Range loop
   (perform the operation for List (I))
end loop;
```

INSTRUCTOR NOTES

ARRAYS CAN ALSO BE USED TO HOLD LISTS THAT GROW AND SHRINK DURING THE EXECUTION OF THE PROGRAM, PROVIDED THAT THE LISTS DO NOT GROW LONGER THAN SOME PREDETERMINED LENGTH.

THE UPPER LIMIT IS USED AS THE SIZE OF THE ARRAY, AND A SEPARATE INTEGER IS USED TO KEEP TRACK OF THE LAST POSITION WHICH IS CURRENTLY CONSIDERED PART OF THE LIST, I.E., THE CURRENT LENGTH OF THE LIST.

BECAUSE BOTH THE ARRAY ELEMENTS AND THE CURRENT LENGTH ARE NEEDED TO COMPLETELY DESCRIBE THE LIST, THE TYPE List_Type IS DEFINED AS A <u>RECORD</u> WITH AN ARRAY COMPONENT AND AN INTEGER COMPONENT. THEN A DECLARATION OF List AS A VARIABLE OF TYPE List_Type AUTOMATICALLY PROVIDES <u>BOTH</u> COMPONENTS CONSTITUTING THE COMPLETE DESCRIPTION OF A LIST.

(THE DEFINITION OF TYPE Storage_Space IS NEEDED BECAUSE A RECORD COMPONENT MAY NOT BELONG TO AN ANONYMOUS ARRAY TYPE. THAT IS, THE FOLLOWING COMPONENT DECLARATION IS <u>ILLEGAL</u>:

```
List_Elements : array (1 .. Maximum_List_Size) of Item_Type;
```

THIS IS A CHANGE FROM THE JULY 1980 PROPOSED Ada STANDARD.)

ONE-DIMENSIONAL ARRAYS AS VARIABLE-LENGTH LISTS

KEEP TRACK OF THE LAST POSITION CURRENTLY CONSIDERED PART OF THE LIST.

THERE IS STILL A MAXIMUM LIST LENGTH.

```
type Storage_Space is array (1 .. Maximum_List_Size) of Item_Type;

type List_Type is
   record
      List_Elements  : Storage_Space;
      Current_Length : Integer range 0 .. Maximum_List_Size;
   end record;

List : List_Type;
```

INSTRUCTOR NOTES

WHEN LISTS ARE IMPLEMENTED IN THIS WAY, REFERENCES TO List(I) ARE REPLACED BY REFERENCES TO List.List_Elements(I). FURTHERMORE, THE LOOP PERFORMING SOME OPERATION FOR EACH ELEMENT IN THE LIST DOES NOT END WITH THE LAST INDEX POSITION OF THE ARRAY, BUT WITH THE LAST INDEX POSITION CURRENTLY IN USE.

IN ADDITION, CERTAIN OPERATIONS THAT ARE IMPOSSIBLE OR UNINTERESTING FOR FIXED-LENGTH LISTS ARE QUITE REASONABLE FOR VARIABLE-LENGTH LISTS:

- DETERMINING THE CURRENT LENGTH OF THE LIST
- CHANGING THE LENGTH OF A LIST WITH ROOM TO EXPAND BY INSERTING A SPECIFIED VALUE AT A SPECIFIED POSITION (IMPLEMENTATION ON NEXT SLIDE)
- CHANGING THE LENGTH OF A RUN-EMPTY LIST BY REMOVING THE VALUE AT A SPECIFIED POSITION (IMPLEMENTATION TWO SLIDES AHEAD)

ONE-DIMENSIONAL ARRAYS AS VARIABLE-LENGTH LISTS (CONTINUED)

PERFORMING SOME OPERATION FOR EACH VALUE IN THE LIST:

```
for I in 1 .. List.Current_Length loop
   (perform the operation for List.List_Elements (I))
end loop;
```

OTHER OPERATIONS WHEN THE LIST LENGTH IS VARIABLE:

- CURRENT LENGTH OF List: List.Current_Length
- INSERT VALUE X BEFORE THE Nth ITEM
  (assuming List.Current_Length < Maximum_List_Size)
- DELETE THE ITEM OF POSITION N
  (assuming 1 <= N <= List.Current_Length)

TO INSERT A NEW VALUE BEFORE THE NTH ELEMENT OF A LIST, EVERY LIST COMPONENT IN POSITIONS N AND HIGHER MUST BE SHIFTED TO THE NEXT HIGHER ARRAY POSITION TO MAKE ROOM. THEN THE NEW VALUE IS PLACED IN POSITION N AND THE CURRENT LENGTH IS INCREMENTED.

IN MOST LANGUAGES SHIFTING MUST BE ACCOMPLISHED BY A LOOP THAT WORKS BACKWARD FROM THE END OF THE LIST, ELEMENT-BY-ELEMENT. (IT WORKS BACKWARDS TO AVOID OVERWRITING DATA.) AS THE TOP SEQUENCE ON THE SLIDE SHOWS, IT IS POSSIBLE TO DO THIS IN ADA. (THE WORD REVERSE PRECEDING THE RANGE IN A FOR-LOOP SPECIFIES THAT THE LOOP PARAMETER IS TO ASSUME THE VALUES OF THE RANGE IN REVERSE ORDER.)

HOWEVER, AS THE MIDDLE SEQUENCE ON THE SLIDE SHOWS, THERE IS A MORE DIRECT WAY TO SHIFT ELEMENTS OF AN ARRAY, BY ASSIGNING ONE SLICE OF THE ARRAY TO ANOTHER SLICE. THE VALUES IN POSITIONS N THROUGH List.Current_Length ARE PICKED UP SIMULTANEOUSLY AND DROPPED INTO POSITIONS N + 1 THROUGH Current_Length + 1.

FINALLY, THE BOTTOM SEQUENCE SHOWS AN ALTERNATIVE FORMULATION USING BOTH CATENATION AND SLICES TO SHIFT THE OLD ELEMENTS AND INSERT THE NEW ONE IN ONE STATEMENT. POSITIONS 1 THROUGH List.Current_Length+1 ARE REPLACED BY A NEW ARRAY VALUE CONSISTING OF THE OLD VALUES BEFORE THE INSERTION POINT, THE INSERTED VALUE, AND THE OLD VALUES AFTER THE INSERTION POINT.

INSERT VALUE X BEFORE THE Nth ITEM

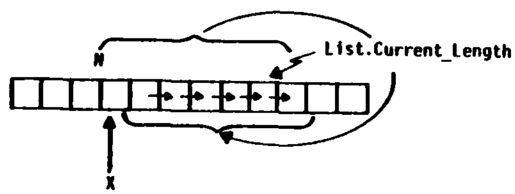
(assuming List.Current_Length < Maximum_List_Size)

THE OLD FASHIONED WAY:

```
for I in reverse N .. List.Current_Length loop
   List.Elements (I + 1) := List.Elements (I);
end loop;
List.Current_Length := List.Current_Length + 1;
List.Elements (N) := X;
```

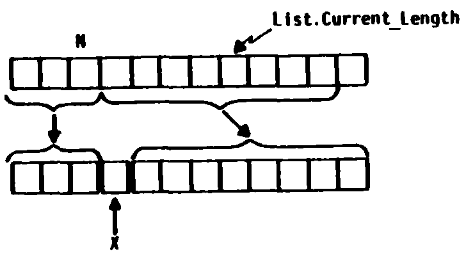
THE NEW, IMPROVED ADA WAY:

```
List.Elements (N + 1 .. List.Current_Length + 1) :=
   List.Elements (N .. List.Current_Length);
List.Current_Length := List.Current_Length + 1;
List.Elements (N) := X;
```



ALTERNATIVELY:

```
List.Elements (1 .. List.Current_Length + 1) :=
   List.Elements (1 .. N-1) & X & List.Elements (N .. List.Current_Length);
List.Current_Length := List.Current_Length + 1;
```



DELETION OF THE ITEM AT POSITION N CONSISTS OF DECREMENTING THE LENGTH AND SHIFTING LIST ELEMENTS IN HIGHER POSITION TO THE NEXT LOWER ARRAY POSITION.

AGAIN, THERE ARE THREE APPROACHES AVAILABLE:

- THE TRADITIONAL COMPONENT-BY-COMPONENT LOOP

- A SLICE ASSIGNMENT MOVING THE VALUES IN POSITIONS N+1 THROUGH List.Current_Length+1 TO POSITION N THROUGH List.Current_Length

- COMBINED USE OF SLICES AND CATENATION TO BUILD AN ARRAY CONSISTING OF THE VALUES BEFORE THE DELETED ELEMENT AND THE VALUES AFTER THE DELETED ELEMENT, AND PLACE THIS ARRAY VALUE IN POSITIONS 1 THROUGH List.Current_Length

DELETE ITEM AT POSITION N

(assuming $1 \le N \le$ List.Current_Length)

THE OLD FASHIONED WAY:

```
List.Current_Length := List.Current_Length - 1;
for I in N .. List.Current_Length loop
   List.Elements (I) := List.Elements (I+1);
end loop;
```

THE NEW, IMPROVED ADA WAY:

```
List.Current_Length := List.Current_Length - 1;
List.Elements (N .. List.Current_Length) :=
   List.Elements (N+1 .. List.Current_Length + 1);
```

ALTERNATIVELY:

```
List.Current_Length := List.Current_Length - 1;
List.Elements (1 .. List.Current_Length) :=
   List.Elements (1 .. N-1) & List.Elements (N+1 .. List.Current_Length + 1);
```

INSTRUCTOR NOTES

PUSHING IS A SPECIAL CASE OF INSERTING AND POPPING IS A SPECIAL CASE OF DELETING. THESE SPECIAL CASES ARE MORE EFFICIENT THAN THE GENERAL CASE BECAUSE THERE ARE NO ELEMENTS IN HIGHER POSITIONS TO BE MOVED.

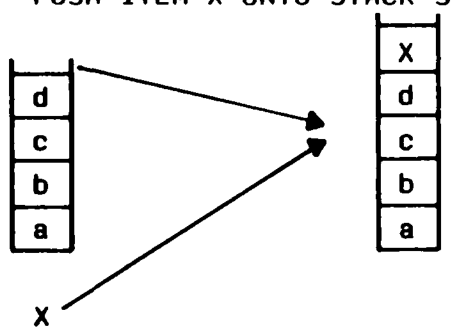
STACKS

A STACK IS A VARIABLE-LENGTH LIST IN WHICH ITEMS ARE ALWAYS INSERTED AND DELETED FROM ONE END.

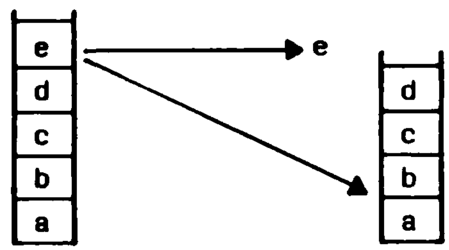
THAT END IS CALLED THE TOP OF THE STACK

OPERATIONS:

- PUSH ITEM X ONTO STACK S:



- POP AN ITEM OFF STACK S:



- TEST WHETHER A STACK IS EMPTY

INSTRUCTOR NOTES

NOTE THAT THE FIRST MESSAGE DOES NOT BEGIN WITH A "[", BUT ALL MESSAGES END WITH "]".

THE NEXT SLIDE HAS AN EXAMPLE.

PROBLEM

WRITE A PROGRAM TO RECEIVE AND PRINT MESSAGES. MESSAGES OF DIFFERING PRIORITIES ARE INTERLEAVED.

THE "[" CHARACTER MEANS THAT THE MESSAGE CURRENTLY BEING RECEIVED IS TO BE INTERRUPTED IN ORDER TO RECEIVE A HIGHER-PRIORITY MESSAGE.

THE "]" CHARACTER INDICATES THE END OF A MESSAGE. THE MESSAGE SHOULD BE PRINTED (WITH INTERVENING HIGHER-PRIORITY MESSAGES REMOVED), AND RECEPTION OF THE MESSAGE INTERRUPTED BY THIS MESSAGE SHOULD BE RESUMED.

ASSUMPTIONS: MESSAGES HAVE $\leq$ 132 CHARACTERS.

NO MORE THAN 10 INTERRUPTED MESSAGES AT ONCE.

MESSAGES ARE RECEIVED FROM STANDARD INPUT AND ARE DISPLAYED ON STANDARD OUTPUT.

INSTRUCTOR NOTES

THE TOP LINE SHOWS A SAMPLE INPUT:

THE CHARACTERS EASYƀTOƀ ARE RECEIVED.

THE [CAUSES PROCESSING OF THIS MESSAGE TO BE SUSPENDED.

THE CHARACTERS HANDLINGƀ ARE RECEIVED AS PART OF A HIGHER-PRIORITY MESSAGE.

THE [CAUSES PROCESSING OF THE MESSAGE TO BE SUSPENDED TOO.

THE HIGHER-PRIORITY MESSAGE ROSESƀAREƀRED. IS RECEIVED IN ITS ENTIRETY.

THE] CAUSES THE MESSAGE TO BE PRINTED AND THE HANDLING ... MESSAGE IS RECEIVED.

THE CHARACTERS AƀMESS ARE RECEIVED. THE MESSAGE SO FAR IS HANDLINGƀAƀMESS.

THE [CAUSES PROCESSING OF THIS MESSAGE TO BE SUSPENDED.

THE HIGHER-PRIORITY MESSAGE VIOLETSƀAREƀBLUE. IS RECEIVED IN ITS ENTIRETY.

THE] CAUSES THE MESSAGE TO BE PRINTED, AND THE HANDLINGƀAƀMESS MESSAGE IS RESUMED.

THE CHARACTERS AGEƀ ARE RECEIVED, YIELDING HANDLINGƀMESSAGEƀIS.

THE] CAUSES THE MESSAGE TO BE PRINTED, AND THE EASYƀTO ... MESSAGE IS RESUMED.

THE CHARACTERS DO. ARE RECEIVED, YIELDING EASYƀTOƀDO.

THE] CAUSES THE MESSAGE TO BE PRINTED. SINCE NO SUSPENDED MESSAGES REMAIN, THE PROGRAM TERMINATES.

EXAMPLE

INPUT:

EASY TO [HANDLING [ROSES ARE RED.] A MESS [VIOLETS ARE BLUE.] AGE IS] DO.]

EASY TO

HANDLING

ROSES ARE RED.

A MESS

VIOLETS ARE BLUE.

AGE IS

DO.

OUTPUT:

ROSES ARE RED.

VIOLETS ARE BLUE.

HANDLING A MESSAGE IS

EASY TO DO.

INSTRUCTOR NOTES

STACKS ARE USEFUL FOR SAVING INFORMATION IN A PARTICULAR ORDER AND RECOVERING IT IN THE REVERSE ORDER (SO THAT THE MOST RECENTLY SAVED INFORMATION IS RECOVERED FIRST).

IN EACH FRAME, THE MESSAGE CURRENTLY BEING PROCESSED IS ON THE LEFT AND THE STACK CONTAINING PARTS OF INTERRUPTED MESSAGES IS ON THE RIGHT.

FRAME 1: THE CHARACTERS EASY TO ARE RECEIVED.

FRAME 2: THE EASY TO MESSAGE IS INTERRUPTED AND PUSHED ON THE STACK, AND THE CHARACTERS HANDLING ARE RECEIVED.

FRAME 3: THE HANDLING MESSAGE IS INTERRUPTED AND PUSHED ON THE STACK, AND THE CHARACTERS ROSES ARE RED. ARE RECEIVED.

FRAME 4: THE ROSES ARE RED. MESSAGE IS PRINTED AND THE TOP MESSAGE IS POPPED OFF THE STACK TO BECOME THE CURRENT MESSAGE AGAIN.

FRAME 5 THE CHARACTERS A MESS ARE RECEIVED AND ADDED TO THE CURRENT MESSAGE.

FRAME 6: THE CURRENT MESSAGE IS INTERRUPTED AND PUSHED ON THE STACK, AND THE CHARACTERS VIOLETS ARE BLUE. ARE RECEIVED

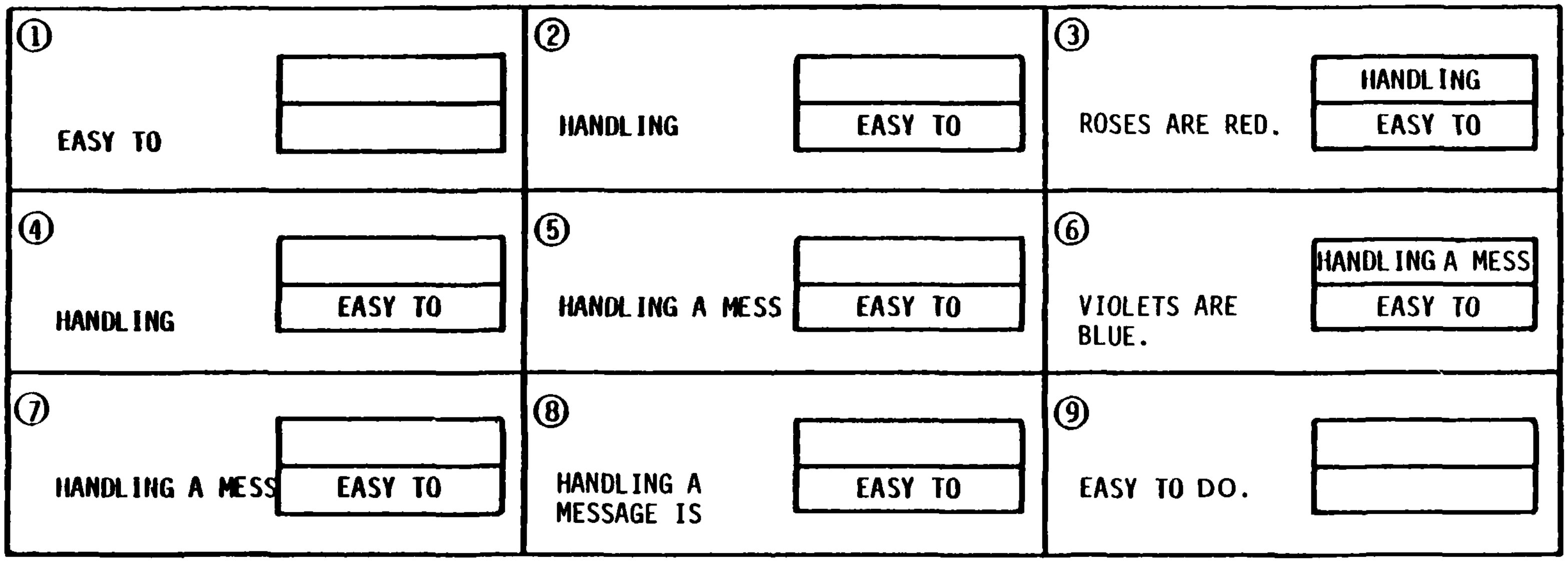
FRAME 7: THE VIOLETS ARE BLUE. MESSAGE IS PRINTED AND THE TOP MESSAGE IS POPPED OFF THE STACK TO BECOME THE CURRENT MESSAGE AGAIN.

FRAME 8: THE CHARACTERS AGE IS ARE RECEIVED AND ADDED TO THE CURRENT MESSAGE.

FRAME 9: THE MESSAGE HANDLING A MESSAGE IS IS PRINTED AND THE TOP MESSAGE IS POPPED OFF THE STACK TO BECOME THE CURRENT MESSAGE AGAIN. ONCE THAT MESSAGE IS COMPLETED AND PRINTED, THERE ARE NO SUSPENDED MESSAGES LEFT ON THE STACK, SO THE PROGRAM TERMINATES.

SOLUTION

- USE A STACK OF MESSAGES
- WHEN RECEPTION OF A MESSAGE IS INTERRUPTED, PUSH THE PARTIALLY COMPLETED MESSAGE ONTO THE STACK.
- WHEN AN INTERRUPTING MESSAGE IS COMPLETE, POP THE PARTIAL MESSAGE IT INTERRUPTED AND RESUME PROCESSING OF THAT MESSAGE.



INSTRUCTOR NOTES

THE PROGRAM WILL BE DESCRIBED IN TERMS OF STACK OPERATIONS, THEN WE'LL SHOW HOW TO TRANSLATE THOSE OPERATIONS INTO ADA. STACK OPERATIONS ARE ENCLOSED IN BOXES.

Message_Type IS ESSENTIALLY A VARIABLE-LENGTH LIST OF CHARACTERS. (STRING IS AN ARRAY OF CHARACTERS.)

PSEUDO-Ada SOLUTION

```
with Text_IO;

procedure Print_Messages is

   Max_Message_Size : constant := 132;
   Max_Stack_Size   : constant := 10;

   Start_Character : constant Character := '[';
   End_Character   : constant Character := ']';

   type Message_Type is
      record
         Contents : String (1 .. Max_Message_Size);
         Length   : Integer range 0 .. Max_Message_Size;
      end record;

   type Stack_Type is ???;

   Current_Message       : Message_Type := ((1 .. Max_Message_Size =>  ' '), 0);

   Partial_Message_Stack : Stack_Type := (Empty_Stack) ;

   Next_Character        : Character;

begin -- Print_Messages

   statements on next slide

end Print_Messages;
```

INSTRUCTOR NOTES

THE ENTIRE SEQUENCE OF STATEMENTS CONSISTS OF A LOOP. THE LOOP IS EXECUTED ONCE FOR EACH CHARACTER. IT IS EXITED FROM THE SECOND BRANCH OF THE CASE STATEMENT, WHEN THE End_Character IS ENCOUNTERED WITH AN EMPTY STACK.

THE NORMAL PROCESSING OF A CHARACTER IS PERFORMED BY THE THIRD (WHEN OTHERS) BRANCH OF THE CASE STATEMENT. IT CONSISTS OF ADDING THE CHARACTER JUST READ TO THE END OF THE CURRENT MESSAGE. THIS INVOLVES UPDATING THE LENGTH OF THE CURRENT MESSAGE AND INSERTING THE CHARACTER.

WHEN THE Start_Character IS ENCOUNTERED, THE Message_Type VALUE IN Current_Message IS PUSHED ONTO THE STACK. THEN Current_Message.Length IS SET TO ZERO, WHICH HAS THE EFFECT OF REPLACING THE OLD CONTENTS OF Current_Message WITH A MESSAGE OF LENGTH ZERO. THIS IS IN PREPARATION FOR THE RECEIPT OF A HIGHER-PRIORITY MESSAGE.

WHEN THE End_Character IS ENCOUNTERED, THE Current_Message IS PRINTED. Current_Message.Length IS USED TO DETERMINE THE SLICE OF Current_Message.Contents THAT ACTUALLY CONTAINS THE MESSAGE. IF THE STACK IS EMPTY, THE LOOP IS EXITED, TERMINATING THE PROGRAM. OTHERWISE, THE Message_Type OBJECT MOST RECENTLY PUSHED ONTO THE STACK IS POPPED OFF IT AGAIN, INTO Current_Message. IT REPLACES THE MESSAGE THAT WAS JUST PRINTED.

(THE PROGRAM DOES NOT CHECK THAT THE INPUT OBEYS THE LIMITS ON MESSAGE LENGTH AND NUMBER OF MESSAGES INTERRUPTED AT ONCE. THE PROGRAM WILL NOT WORK CORRECTLY IF THE INPUT VIOLATES THESE ASSUMPTIONS.)

PSEUDO-Ada SOLUTION (Continued)

```
with Text_IO;

procedure Print_Messages is

   declarations on previous slide

begin -- Print_Messages

   loop

      Text_IO.Get (Next_Character);
      case Next_Character is
         when Start_Character =>

            push Current_Message onto Partial_Message_Stack ;

            Current_Message.Length := 0;
         when End_Character =>
            Text_IO.Put (Current_Message.Contents (1 .. Current_Message.Length));
            Text_IO.New_Line;

            exit when Partial_Message_Stack is empty ;  -->>> LOOP EXIT <<<--

            pop Partial_Message_Stack into Current_Message ;

         when others =>
            Current_Message.Length := Current_Message.Length + 1;
            Current_Message.Contents (Current_Message.Length) := Next_Character;
      end case;

   end loop;

end Print_Messages;
```

THESE OPERATIONS ARE ESSENTIALLY A SPECIAL CASE OF THE VARIABLE-LENGTH LIST OPERATIONS. (THE TOP COMPONENT PLAYS THE SAME ROLE AS THE Current_Length COMPONENT PLAYED EARLIER). BECAUSE THE POSITION OPERATED ON IS ALWAYS THE LAST POSITION IN THE LIST, THERE IS NO NEED TO SHIFT ITEMS IN HIGHER POSITIONS.

A NOTE FOR THE EFFICIENCY-MINDED:

THE CONTENTS OF Current_Message ARE IRRELEVANT JUST AFTER THEY HAVE BEEN PUSHED ON THE STACK, BECAUSE THEY ARE IMMEDIATELY REPLACED BY A MESSAGE OF LENGTH ZERO. THIS MAKES THE FOLLOWING IMPROVEMENT POSSIBLE:

ALL OPERATIONS ON Current_Message CAN BE REPLACED BY OPERATIONS ON Partial_Message_Stack.Elements (Partial_Message_Stack.Top+1), THE CELL JUST ABOVE THE TOP OF THE STACK THEN THE ASSIGNMENT Partial_Message_Stack.Top := Partial_Message_Stack.Top+1 HAS THE EFFECT OF PUSHING THE CURRENT MESSAGE ONTO THE STACK, WITHOUT THE NEED TO COPY A LARGE Message_Type OBJECT. THIS CAUSES THE NEXT EMPTY STACK CELL TO START ASSUMING THE ROLE OF Current_Message, BUT THAT'S OKAY BECAUSE THE CONTENTS OF Current_Message ARE IRRELEVANT AT THIS POINT. SIMILARLY, THE ASSIGNMENT Partial_Message_Stack.Top := Partial_Message_Stack.Top-1 CAUSES THE STACK CELL THAT WAS THE TOP OF THE STACK TO ASSUME THE ROLE OF Current_Message INSTEAD. THIS ACHIEVES THE EFFECT OF POPPING INTO Current_Message WITHOUT COPYING A LARGE Message_Type OBJECT.

SINCE ONE OF THE STACK CELLS IS NOW USED TO HOLD THE CURRENT MESSAGE, THE SIZE OF THE STACK MUST BE INCREASED BY ONE TO ACCOMMODATE THE SAME NUMBER OF INTERRUPTED MESSAGES.

IMPLEMENTATION OF STACKS

```
type Message_Array is array (1 .. Max_Stack_Size) of Message_Type;

type Stack_Type is
   record
      Elements : Message_Array;
      Top      : Integer range 0 .. Max_Stack_Size := 0;
   end record;
```

Partial_Message_Stack : Stack_Type := (empty stack);

```
   initialization to an empty stack is automatic because
   the default initial value of Partial_Message_Stack.Top is 0;
```

Partial_Message_Stack is empty

```
   Partial_Message_Stack.Top = 0
```

push Current_Message onto Partial_Message_Stack

```
   Partial_Message_Stack.Top := Partial_Message_Stack.Top + 1;
   Partial_Message_Stack.Elements (Partial_Message_Stack.Top) :=
      Current_Message;
```

pop Partial_Message_Stack into Current_Message

```
   (assuming Partial_Message_Stack is not empty)

   Current_Message :=
      Partial_Message_Stack.Elements (Partial_Message_Stack.Top);
   Partial_Message_Stack.Top := Partial_Message_Stack.Top - 1;
```

INSTRUCTOR NOTES

INSERTION IS ONLY ALLOWED WHEN THE QUEUE IS NOT FULL. REMOVAL IS ONLY ALLOWED WHEN THE QUEUE IS NOT EMPTY.

QUEUES

A QUEUE IS A VARIABLE-LENGTH LIST IN WHICH ITEMS ARE ALWAYS INSERTED AT ONE END (THE BACK) AND DELETED FROM THE OTHER END (THE FRONT).

<u>OPERATIONS:</u>

INSERT ITEM X AT THE BACK OF THE QUEUE.

EXAMINE THE ITEM AT THE FRONT OF THE QUEUE.

REMOVE THE FRONT ITEM IN THE QUEUE.

TEST WHETHER THE QUEUE IS EMPTY.

TEST WHETHER THE QUEUE IS FULL.

TO DEMONSTRATE THE USE OF QUEUES, WE SOLVE THE FOLLOWING INVENTORY MANAGEMENT PROBLEM.

WE ARE GIVEN AN INPUT FILE CONSISTING OF TRANSACTIONS REPORTING TWO KINDS OF EVENTS -- A SPECIFIED CUSTOMER ORDERING A SPECIFIED NUMBER OF ITEMS, AND A SPECIFIED NUMBER OF ITEMS ARRIVING AT THE WAREHOUSE. THERE IS ONLY ONE KIND OF ITEM.

WE MUST PRODUCE AN OUTPUT FILE THAT REPORTS THE PLACEMENT OF ORDERS, THE ARRIVAL OF ITEMS AT THE WAREHOUSE, AND THE DELIVERY OF ITEMS TO CUSTOMERS, IN THE ORDER IN WHICH THOSE EVENTS TAKE PLACE.

ORDERS ARE FILLED IN THE ORDER WHICH THEY ARRIVE, PROVIDED THAT THERE IS SUFFICIENT INVENTORY IN THE WAREHOUSE. IF THERE IS NOT SUFFICIENT INVENTORY TO FULFILL THE OLDEST PENDING ORDER, ALL CUSTOMERS MUST WAIT FOR MORE INVENTORY TO ARRIVE. IF TEN CUSTOMERS ARE WAITING FOR THEIR ORDERS TO BE FILLED, ADDITIONAL ORDERS ARE NOT ACCEPTED. THE OUTPUT SHOULD REPORT BOTH ORDERS THAT WERE ACCEPTED AND ORDERS THAT WERE NOT.

THE INITIAL INVENTORY IS ZERO.

PROBLEM

INPUT: TRANSACTION RECORDS

O 10 27	(CUSTOMER 27 ORDERS 10 ITEMS.)
R 3	(3 ITEMS RECEIVED FOR SHIPMENTS)

OUTPUT: LOG OF EVENTS

PLACEMENT OF AN ORDER
ARRIVAL OF ITEMS
SHIPMENT OF ITEMS

POLICY:

CUSTOMERS SERVED ON A *FIRST-COME*, FIRST-SERVED BASIS.
NO CUSTOMER IS SERVED UNTIL HIS ORDER CAN BE FILLED IN ITS ENTIRETY.
ORDERS RECEIVED WITH TEN CUSTOMERS WAITING ARE IGNORED.

CONSEQUENCE:

IF CUSTOMER A ORDERS 10 ITEMS, THEN CUSTOMER B ORDERS 3 ITEMS, AND 5 ITEMS ARE ON HAND, THEN BOTH A AND B MUST WAIT (OR B'S ORDER MUST BE IGNORED).

INSTRUCTOR NOTES

TO KEEP THE MAIN PROGRAM SHORT AND UNCLUTTERED, WE ASSUME THAT ALL INPUT AND OUTPUT IS HANDLED BY A PACKAGE NAMED Interface_Package. Interface_Package IS DEFINED IN TERMS OF THE Text_IO PACKAGE AND PROVIDES A FUNCTION TO INDICATE End_Of_File, A SUBPROGRAM TO READ A TRANSACTION, AND FOUR SUBPROGRAMS TO PRINT REPORTS OF EVENTS.

TYPES REFERRED TO BOTH IN THE Interface_Package AND IN THE INVENTORY MANAGEMENT PROCEDURE ARE DEFINED IN ANOTHER PACKAGE, Type_Definition_Package. THUS Interface_Package USES Type_Definition_Package and Text_IO, WHILE THE INVENTORY MANAGEMENT PROCEDURE USES Type_Definition_Package AND Interface_Package.

THE Type_Definition_Package DEFINES INTEGER TYPES TO REPRESENT COUNTS OF INVENTORY ITEMS AND CUSTOMER NUMBERS, A RECORD TYPE TO REPRESENT AN ORDER CONSISTING OF A CUSTOMER NUMBER AND A NUMBER OF ITEMS, AND AN ENUMERATION TYPE TO INDICATE WHICH KIND OF TRANSACTION HAS BEEN READ IN.

SOLUTION

USE A QUEUE OF ORDERS.

ASSUME THE FOLLOWING "SERVICE PACKAGES":

```
package Type_Definition_Package is
   type Item_Count_Type is range 0 .. 1000;
   type Customer_Number_Type is range 0 .. 1000;
   type Order_Type is
      record
         Customer_Number_Part : Customer_Number_Type;
         Item_Count_Part      : Item_Count_Type;
      end record;
   type Transaction_Type is (Order_Transaction, Items_Received_Transaction);
end Type_Definition_Package;

with Text_IO;
with Type_Definition_Package; use Type_Definition_Package;

package Interface_Package is
   function End_of_File return Boolean renames Text_IO.End_Of_File;
   procedure Get_Transaction (Transaction              : out Transaction_Type;
                              Item_Count               : out Item_Count_Type;
                              Customer_Number          : out Customer_Number_Type);
   procedure Report_Order_Placed (Customer               : in Customer_Number_Type;
                                  Amount_Ordered         : in Item_Count_Type;
                                  New_Inventory          : in Item_Count_Type);
   procedure Report_Order_Ignored (Customer              : in Customer_Number_Type;
                                   Amount_Needed         : in Item_Count_Type);
   procedure Report_Items_Received (Amount_Received      : in Item_Count_Type;
                                    New_Inventory        : in Item_Count_Type);
   procedure Report_Item_Sent (Customer                  : in Customer_Number_Type;
                               Amount_Sent               : in Item_Count_Type;
                               New_Inventory             : in Item_Count_Type);
end Interface_Package;
```

INSTRUCTOR NOTES

THE MAIN PROCEDURE WILL BE DESCRIBED IN TERMS OF QUEUE OPERATIONS, AND LATER SLIDES WILL EXPLAIN THE TRANSLATION OF THESE OPERATIONS INTO ADA.

QUEUE OPERATIONS ARE ENCLOSED IN BOXES.

PSEUDO-Ada SOLUTION

```
with Type_Definition_Package, Interface_Package;
use Type_Definition_Package, Interface_Package;

procedure Manage_Inventory is

   Queue_Size : constant := 10;
   type Queue_Type is ???;

   Inventory         : Item_Count_Type := 0;

   Queue             : Queue_Type := (empty queue);

   Customer_Number   : Customer_Number_Type;
   Item_Count        : Item_Count_Type;
   Transaction       : Transaction_Type;

begin -- Manage_Inventory

   statements on next slide

end Manage_Inventory;
```

INSTRUCTOR NOTES

THE MAIN PROGRAM CONSISTS OF A WHILE LOOP REPEATED ONCE FOR EACH TRANSACTION IN THE INPUT FILE.

EACH REPETITION OF THE LOOP CALLS Get_Transaction TO READ A TRANSACTION, EXECUTES A CASE STATEMENT THAT PROCESSES AND REPORTS THE TRANSACTION, AND THEN EXECUTES A NESTED WHILE-LOOP THAT FILLS ALL ORDERS THAT CAN BE FILLED AS A RESULT OF THE TRANSACTION.

A TRANSACTION CAN ENABLE ORDERS TO BE FILLED EITHER BY ADDING ENOUGH INVENTORY TO ALLOW THE OLDEST ORDER IN THE QUEUE TO BE FILLED OR BY ADDING A FILLABLE ORDER TO PREVIOUSLY EMPTY QUEUE.

THE CASE STATEMENT PROCESSES ORDERS BY REPORTING A TURNED AWAY CUSTOMER IF THE QUEUE IS FULL, OR BY REPORTING AN ACCEPTED ORDER AND ADDING THE ORDER TO THE BACK OF THE QUEUE IF THERE IS ROOM. THE CASE STATEMENT PROCESSES THE ARRIVAL OF MORE INVENTORY BY ADDING THE SPECIFIED AMOUNT TO THE VARIABLE Inventory AND REPORTING THE EVENT.

THE INNER WHILE LOOP IS REPEATED AS LONG AS THERE IS A FILLABLE ORDER AT THE FRONT OF THE QUEUE. FILLING AN ORDER CONSISTS OF DECREASING THE VARIABLE Inventory BY THE AMOUNT OF THE ORDER, REPORTING THE EVENT, AND REMOVING THE ORDER FROM THE QUEUE. THE WHILE CONDITION IS WRITTEN AS A SHORT CIRCUIT CONTROL FORM BECAUSE IT IS MEANINGLESS TO REFER TO THE FIRST ORDER OF QUEUE WHEN QUEUE IS EMPTY.

```
with Type_Definition_Package, Interface_Package;
use Type_Definition_Package, Interface_Package;

procedure Manage_Inventory is
   declarations on previous slide
begin -- Manage_Inventory
   while not End_of_File loop
      Get_Transaction (Transaction, Item_Count, Customer_Number);
      case Transaction is
         when Order_Transaction =>
            if Queue is full then
               Report_Order_Ignored (Customer_Number, Item_Count);
            else
               Report_Order_Placed (Customer_Number, Item_Count);
               insert Order_Type'(Customer_Number, Item_Count) in Queue ;
            end if;
         when Items_Received_Transaction =>
            Inventory := Inventory + Item_Count;
            Report_Items_Received (Item_Count, Inventory);
      end case;
      while not Queue is empty and then
            Inventory >= first order in Queue .Item_Count_Part loop
         Inventory := Inventory - first order in Queue .Item_Count_Part;
         Report_Items_Sent ( first order in Queue .Customer.Number_Part,
                             first order in Queue .Item_Count_Part,
                             Inventory)
        remove first item from Queue ;
      end loop;
   end loop;
end Manage_Inventory;
```

INSTRUCTOR NOTES

PERHAPS THE MOST OBVIOUS IMPLEMENTATION OF A QUEUE IS AS A VARIABLE-LENGTH LIST, WHERE THE FIRST ITEM IN THE LIST IS AT THE FRONT OF THE QUEUE AND THE LAST ITEM IN THE LIST IS AT THE BACK OF THE QUEUE.

IMPLEMENTATION OF QUEUES

QUEUE:

```
type Order_List is array (Integer range <>) of Order_Type;

type Queue_Type is
   record
      Orders         : Order_List (1 .. Queue_Size);
      Current_Length : Integer range 0 .. Queue_Size := 0;
   end record;
```

OPERATIONS:

<u>Queue : Queue Type := (empty queue);</u>

INITIALIZATION TO AN EMPTY QUEUE IS AUTOMATIC BECAUSE THE DEFAULT INITIAL VALUE OF Queue.Current_Length IS 0.

<u>Queue IS EMPTY</u>

Queue.Current_Length = 0

<u>Queue IS FULL</u>

Queue.Current_Length = Queue_Size

INSTRUCTOR NOTES

SINCE Queue.Orders (1) REPRESENTS THE FIRST ORDER IN QUEUE, AND AN ORDER IS REPRESENTED BY A RECORD, Queue.Orders (1).Item_Count_Part NAMES THE AMOUNT OF THE OLDEST ORDER AND Queue.Orders (1).Customer_Number_Part NAMES THE CUSTOMER WHO PLACED THAT ORDER.

INSERTION IS A FAST OPERATION BECAUSE THERE ARE NO LIST ELEMENTS IN HIGHER-NUMBERED POSITIONS THAT MUST BE SHIFTED TO MAKE ROOM. REMOVAL IS A RELATIVELY SLOW OPERATION BECAUSE EVERY LIST ELEMENT MUST BE SHIFTED DOWNWARDS WHEN THE FIRST ELEMENT IS REMOVED.

THE QUEUE OPERATIONS ARE INTENDED AS AN IN-CLASS EXERCISE:

```
ANSWERS:     FIRST ORDER  : Queue.Orders(1)
             INSERT ORDER :
                   Queue.Current_Length := Queue.Current_Length + 1;
                   Queue.Orders (Queue.Current_Length) := X;
             REMOVE ORDER :
                   Queue.Orders   (1 .. Queue.Current_Length - 1) :=
                      Queue.Orders (2 .. Queue.Current_Length);
                   Queue.Current_Length := Queue.Current_Length - 1;
```

IMPLEMENTATION OF QUEUES (CONTINUED)

FIRST ORDER IN Queue

```
Queue.Orders (1)      -- Queue.Orders (1).Item_Count_Part
                      -- Queue.Orders (1).Customer_Number_Part
```

INSERT ORDER X IN Queue (ASSUMING Queue IS NOT FULL)

REMOVE FIRST ITEM FROM Queue (ASSUMING Queue IS NOT EMPTY)

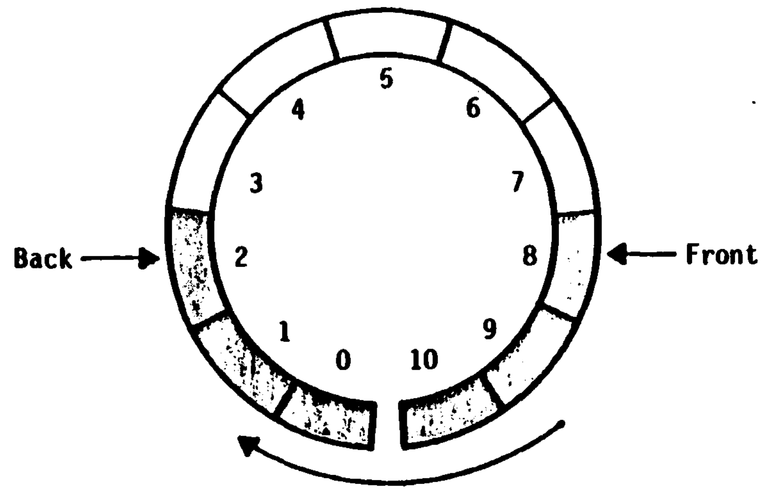
A MORE EFFICIENT REPRESENTATION OF A QUEUE IS AS A CIRCULAR LIST.

IN A CIRCULAR LIST, AN ARRAY IS VIEWED AS HAVING BEEN BENT TO FORM A CIRCLE, SO THAT THE LAST POSITION IN THE LIST IS FOLLOWED IMMEDIATELY BY THE FIRST POSITION.

WHEN AN ITEM IS ADDED TO THE QUEUE, THE BACK MOVES ONE POSITION CLOCKWISE TO EXTEND THE QUEUE. WHEN AN ITEM IS REMOVED FROM THE QUEUE, THE FRONT MOVES ONE POSITION CLOCKWISE TO SHRINK THE QUEUE. OVER TIME, THE FILLED PART OF THE CIRCLE SEEMS TO TRAVEL CLOCKWISE, AS INDICATED BY THE ARROW.

IF THE POSITIONS IN THE ARRAY ARE NUMBERED 0 TO 10, THE FORMULA $(I+1) \bmod 11$ ALWAYS GIVES THE POSITION AFTER POSITION I: WHEN $0 \leq I \leq 9$, $(I+1) \bmod 11 = I+1$; WHEN $I = 10$, $(I+1) \bmod 11 = 11 \bmod 11 = 0$.

CIRCULAR LISTS



ITEMS ARE ADDED TO BACK OF LIST, REMOVED FROM FRONT OF LIST.

INDEX AFTER I IN THIS CIRCULAR LIST: (I+1) MOD 11

INSTRUCTOR NOTES

AN ARRAY WITH ELEMENTS NUMBERED 0 TO 10 HAS 11 ELEMENTS. IN GENERAL, A CIRCULAR LIST MUST HAVE ONE MORE ELEMENT THAN THE MAXIMUM LIST SIZE IT IS TO HOLD. THE EXTRA ELEMENT IS NEEDED TO DISTINGUISH BETWEEN A FULL QUEUE AND AN EMPTY QUEUE.

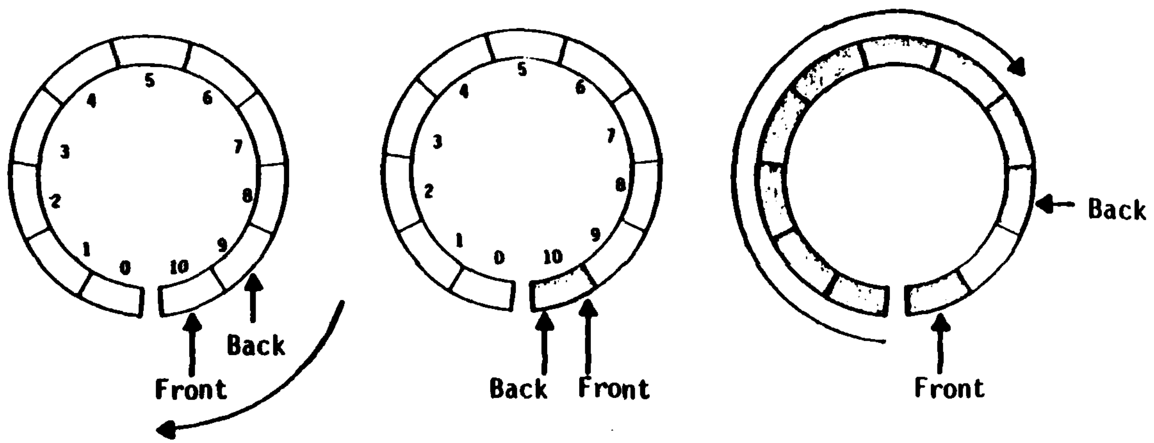
IN AN EMPTY QUEUE, THE BACK POSITION IS ONE BEHIND THE FRONT POSITION. THEN INSERTION OF A SINGLE ELEMENT ADVANCES THE BACK POINTER TO EQUAL THE FRONT POINTER.

IF IT WERE NOT FOR THE EXTRA ELEMENT, A FULL QUEUE WOULD ALSO HAVE THE BACK POSITION ONE BEHIND THE FRONT POSITION. GIVEN THE EXTRA ELEMENT, THE BACK POSITION IS TWO BEHIND THE FRONT POSITION INSTEAD.

CIRCULAR LIST REPRESENTATION OF EMPTY AND FULL QUEUES

AN 11-ELEMENT CIRCULAR LIST IS USED TO REPRESENT A QUEUE WITH A CAPACITY OF 10.

AT LEAST ONE ELEMENT IS ALWAYS KEPT EMPTY.



EMPTY QUEUE
(Front = (Back+1)Mod 11)

QUEUE WITH ONE ITEM
(Front = Back)

FULL QUEUE
(Front = (Back+2) Mod 11)

INSTRUCTOR NOTES

THE TESTS FOR EMPTY AND FULL QUEUES ARE AS EXPLAINED ON THE PREVIOUS SLIDE.

NO DATA IN THE QUEUE HAS TO BE SHIFTED FOR EITHER INSERTION OR REMOVAL. ONLY THE FRONT AND BACK POINTERS MOVE.

ANSWERS:

```
Queue is full
   Queue.Front = (Queue.Back + 2) mod (Queue_Size + 1)


Remove first item from Queue (assuming Queue is not empty)
   Order := Queue.Orders (Queue.Front);
   Queue.Front := (Queue.Front + 1) mod (Queue_Size + 1);
```

IMPLEMENTATION OF A QUEUE AS A CIRCULAR LIST

```
type Order_List is array (Integer range <>) of Order_Type; -- Order_Type DECLARED ON 2-22
subtype Queue_Index_Subtype is Integer range 0 .. Queue_Size;

type Queue_Type is
   record
      Orders : Order_List (Queue_Index_Subtype);
      Front  : Queue_Index_Subtype := 1;
      Back   : Queue_Index_Subtype := 0;
   end record;
```

<u>Queue: Queue_Type := (empty queue);</u>
initialization to an empty queue is automatic because the default initial value of Queue.Front is one more than the default initial value of Queue.Back.

<u>Queue is empty</u>

```
Queue.Front = (Queue.Back + 1) mod (Queue_Size + 1)
```

<u>Queue is full</u>

<u>First order in Queue</u>

```
Queue.Orders (Front)
```

<u>Insert order X in Queue (assuming Queue is not full)</u>

```
Queue.Back := (Queue.Back + 1) mod (Queue_Size + 1);
Queue.Orders (Queue.Back) := X;
```

<u>Remove first item from Queue (assuming Queue is not empty)</u>

INSTRUCTOR NOTES

THERE ARE MANY WAYS OF EXPRESSING THE CONCEPT OF CIRCULAR LISTS. THIS SLIDE IS PRESENTED SO THAT STUDENTS WILL RECOGNIZE SLIGHTLY DIFFERENT VARIATIONS OF THE SAME BASIC IDEA.

NOTE ON BULLET 3: SUPPOSE Queue_Size = 10. For $1 \leq X \leq 10$, $X \bmod 11 = X$, SO $X \bmod 11+1 = X+1$. FOR $X = 11$, $X \bmod 11 = 0$, SO $X \bmod 11+1 = 1$

VARIATIONS YOU MAY SEE ELSEWHERE

- HAVE Front POINT JUST <u>BEYOND</u> THE OLDEST ITEM IN THE QUEUE
  (TESTS FOR EMPTY AND FULL QUEUE AND OPERATIONS EXAMINING AND REMOVING THE FIRST ITEM IN THE QUEUE ARE ADJUSTED ACCORDINGLY.)

- HAVE Back POINT TO THE SPACE THAT IS TO BE FILLED NEXT RATHER THAN THE SPACE THAT HAS BEEN FILLED MOST RECENTLY.
  (TESTS FOR EMPTY AND FULL QUEUE AND THE INSERTION OPERATION ARE ADJUSTED ACCORDINGLY.)

- NUMBER THE INDEX POSITIONS FROM 1 TO Queue_Size + 1 INSTEAD OF 0 TO Queue_Size.
  ((X+1) mod (Queue_Size + 1)) IS CHANGED TO (X mod (Queue_Size + 1) + 1)

THE BASIC IDEA IS THE SAME, BUT CERTAIN EXPRESSIONS ARE DIFFERENT BY $\pm$ 1.

THE EXTRA SLOT IN THE CIRCULAR LIST IS REQUIRED IN ANY CASE.

INSTRUCTOR NOTES

SECTION 5

LINKED LISTS AND RECURSIVE TYPES

INSTRUCTOR NOTES

EACH RECORD IS CALLED A LIST CELL.

LIST CELLS ARE ALLOCATED, SINCE THEY ARE DESIGNATED BY ACCESS VALUES.

A LIST IS REPRESENTED AS AN ACCESS VALUE POINTING TO THE FIRST LIST CELL.

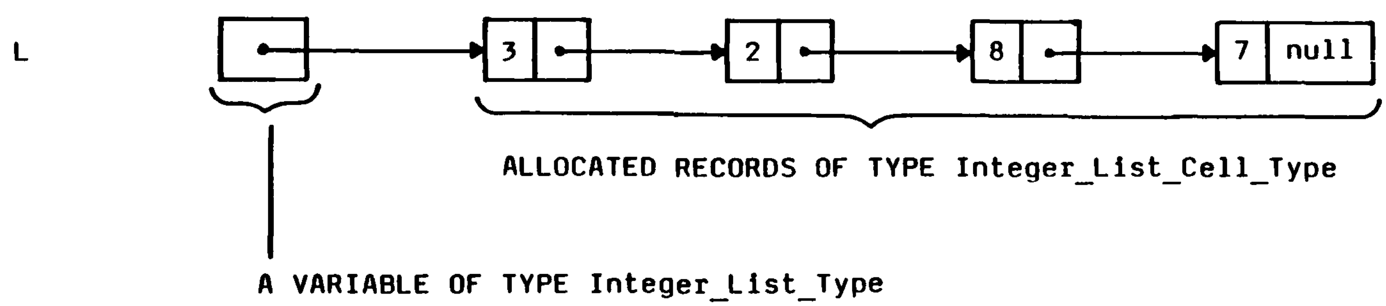
LINKED LISTS

A LIST OF ITEMS BUILT OUT OF RECORDS WITH TWO COMPONENTS:

item	access value pointing to next record (null for last record)

EXAMPLE: A LINKED LIST OF INTEGERS

L HOLDS THE LIST 3, 2, 8, 7:



INSTRUCTOR NOTES

SINCE A LIST IS REPRESENTED AS AN ACCESS VALUE POINTING TO A LIST CELL, AND SINCE THE SECOND COMPONENT OF A LIST CELL IS ALSO AN ACCESS VALUE POINTING TO A LIST CELL, THE SECOND COMPONENT OF THE LIST CELL RECORD SHOULD BELONG TO THE TYPE USED TO REPRESENT LISTS.

HOW DO WE DECLARE THE TYPES USED IN LINKED LISTS?

ATTEMPT 1:

```
type Integer_List_Type is access Integer_List_Cell_Type;

type Integer_List_Cell_Type is
   record
      Item_Part : Integer;
      Link_Part : Integer_List_Type;
   end record;
```

ILLEGAL USE OF Integer_List_Cell_Type BEFORE ITS DECLARATION.

ATTEMPT 2:

```
type Integer_List_Cell_Type is
   record
      Item_Part : Integer;
      Link_Part : Integer_List_Type;
   end record;

type Integer_List_Type is access Integer_List_Cell_Type;
```

ILLEGAL USE OF Integer_List_Type BEFORE ITS DECLARATION!

INSTRUCTOR NOTES

BECAUSE Integer_List_Type IS DEFINED IN TERMS OF Integer_List_Cell_Type AND Integer_List_Cell_Type, IN TURN, IS DEFINED IN TERMS OF Integer_List_Type, Integer_List_Type IS INDIRECTLY DEFINED IN TERMS OF ITSELF. THUS Integer_List_Type IS RECURSIVE.

SIMILARLY, Integer_List_Cell_Type IS RECURSIVE.

THE "SPECIAL MEASURES" THAT MUST BE TAKEN ARE DESCRIBED ON THE NEXT SLIDE.

ADA RECURSIVE TYPES ALWAYS INVOLVE AN ACCESS TYPE AS AT LEAST ONE OF THE LINKS IN THE CIRCULAR DEFINITION.

HOW DO WE DECLARE THE TYPES USED IN LINKED LISTS? (Continued)

PROBLEM: Integer_List_Type AND Integer_List_Cell_Type ARE EACH DEFINED IN TERMS OF THE OTHER.

IN GENERAL, A TYPE WHOSE DECLARATION DEPENDS DIRECTLY OR INDIRECTLY ON ITSELF IS CALLED A <u>RECURSIVE TYPE</u>.

SPECIAL MEASURES MUST BE TAKEN TO ALLOW DECLARATION OF RECURSIVE TYPES WITHOUT REFERRING TO A TYPE BEFORE IT HAS BEEN DECLARED.

AN OBJECT IN A RECURSIVE TYPE CANNOT <u>CONTAIN</u> A COMPONENT OF THE SAME TYPE, BUT IT CAN CONTAIN AN ACCESS VALUE <u>POINTING TO</u> ANOTHER OBJECT OF THE SAME TYPE.

INSTRUCTOR NOTES

AN INCOMPLETE TYPE DECLARATION BREAKS THE CIRCLE BY ALLOWING A TYPE TO BE REFERRED TO BEFORE ITS CONTENTS HAVE BEEN DEFINED.

A FULL TYPE DECLARATION LOOKS LIKE AN ORDINARY TYPE DECLARATION.

SOLUTION: INCOMPLETE TYPE DECLARATIONS

```
type Integer_List_Cell_Type;          -- INCOMPLETE TYPE DECLARATION
type Integer_List_Type is access Integer_List_Cell_Type;

type Integer_List_Cell_Type is
   record
      Item_Part : Integer;
      Link_Part : Integer_List_Type;
   end record;
```

AN <u>INCOMPLETE TYPE DECLARATION</u> HAS THE FORM:

```
type type name;
```

IT INDICATES THAT A <u>FULL DECLARATION</u> FOR THE TYPE WILL FOLLOW LATER, BUT THAT THE TYPE NAME CAN BE USED IN THE MEANTIME IN ACCESS TYPE DECLARATIONS.

-- IT IS AN ERROR IF THE FULL DECLARATION DOES NOT FOLLOW LATER.

-- BETWEEN THE INCOMPLETE DECLARATION AND THE FULL DECLARATION, THE TYPE NAME MAY <u>ONLY</u> BE USED IN AN ACCESS TYPE DECLARATION.

INSTRUCTOR NOTES

STEPS (2) AND (3) MAY INVOLVE THE DECLARATION OF INTERMEDIATE TYPES.

SUMMARY: DECLARING RECURSIVE TYPES

USUAL CASE: A RECORD TYPE CONTAINING ONE OR MORE SUBCOMPONENTS POINTING TO OTHER OBJECTS IN THE SAME RECORD TYPE

(1) INCOMPLETE DECLARATION FOR THE RECORD TYPE

(2) ORDINARY ACCESS TYPE DECLARATION, REFERRING TO THE TYPE DECLARED IN STEP (1)

(3) FULL DECLARATION FOR THE RECORD TYPE, REFERRING TO THE TYPE DECLARED IN STEP (2)

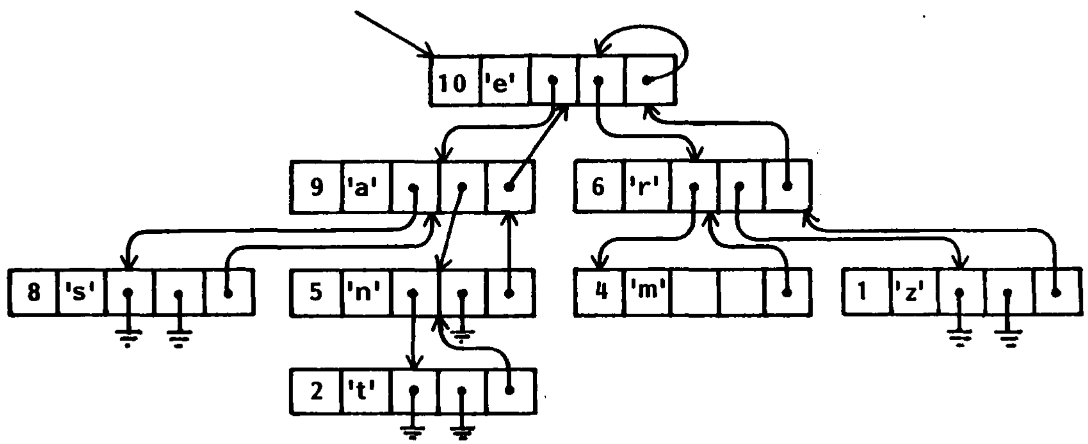
ANSWER:

```
subtype Lower_Case_Alphabet_Type is Character range 'a' .. 'z';
type Character_Frequency_Type;
type Character_Frequency_Tree_Type is access Character_Frequency_Type;
type Character_Frequency_Type is
   record
      Frequency_Count : Natural;
      Char            : Lower_Case_Alphabet_Type;
      Left_Child      : Character_Frequency_Tree_Type;
      Right_Child     : Character_Frequency_Tree_Type;
      Parent          : Character_Frequency_Tree_Type;
   end record;
```

IN-CLASS EXERCISE: RECURSIVE ACCESS TYPES

THIS DIAGRAM REPRESENTS A STRUCTURE TO HOLD CHARACTER FREQUENCY COUNTS.

WRITE THE TYPE DECLARATIONS CORRESPONDING TO THE FOLLOWING DIAGRAM:



INSTRUCTOR NOTES

DETAILS ABOUT Item_Type ARE IRRELEVANT HERE. JUST ASSUME IT'S SOME TYPE THAT HAS BEEN DECLARED EARLIER.

EXPLAIN THE TWO WAYS TO DENOTE A POSITION IN A LIST.

IN THE LOOPS TRAVERSING LISTS, L IS ASSIGNED TO T SO THAT OPERATIONS ON ITEMS IN LIST L DO NOT AFFECT THE VALUE OF L ITSELF.

BE PREPARED TO EXPLAIN HOW T := T.Link_Part WORKS

IMPLEMENTATION OF LIST OPERATIONS USING LINKED LISTS

```
type List_Cell_Type;
type List_Type is access List_Cell_Type;
type List_Cell_Type is
   record
      Item_Part : Item_Type;
      Link_Part : List_Type;
   end record;
```

TWO WAYS TO SPECIFY A POSITION IN A LIST:

-- A POSITIVE NUMBER
-- A POINTER TO A LIST CELL

```
L, Ptr, T : List_Type;
N         : Positive;
X         : Item_Type;
```

- <u>USE [REPLACE] VALUE AT POSITION N IN LIST L</u>:

```
T := L;
for I in 1 .. N - 1 loop
   T := T.Link_Part;
end loop;
X := T.Item_Part;  [T.Item_Part := X;]
```

- <u>USE [REPLACE] VALUE AT POSITION POINTED TO BY Ptr IN LIST L</u>:

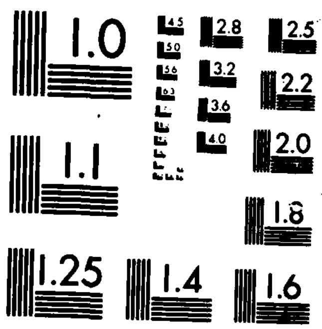
```
X := Ptr.Item_Part;  [Ptr.Item_Part := X;]
```

- <u>PERFORM SOME OPERATION FOR EACH ITEM IN THE LIST</u>:

```
T := L;
while T /= null loop
   (perform the operation for T.Item_Part)
   T := T.Link_Part;
end loop;
```

AD-A165 075 ADA (TRADEMARK) TRAINING CURRICULUM: ADVANCED ADA TOPICS L305 TEACHER'S GUIDE VOLUME 1(U) SOFTECH INC WALTHAM MA 1986 DAAB07-83-C-K506 3/5

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

INSTRUCTOR NOTES

INSERTION AT THE BEGINNING OF A LIST MUST BE HANDLED SPECIALLY BECAUSE THE ACCESS VALUE POINTING TO THE NEW CELL IS PLACED IN THE VARIABLE L RATHER THAN THE Link_Part OF SOME LIST CELL.

INSERTION BEFORE THE LIST CELL POINTED TO BY P REQUIRES FINDING THE LIST CELL WHOSE Link_Part POINTS TO THE SAME CELL AS P, SO THAT IT CAN BE MADE TO POINT TO THE NEW CELL INSTEAD. THIS IS HARD TO DO BECAUSE LINKS GO IN THE OPPOSITE DIRECTION. THE FOLLOWING APPROACH ACCOMPLISHES THIS, BUT ELIMINATES ANY ADVANTAGE OF USING ACCESS VALUES INSTEAD OF NUMBERS TO DENOTE LIST POSITIONS:

```
if Ptr = L then  -- Ptr POINTS TO FIRST CELL
   L := new List_Cell_Type'(X, L);
else
   T := L;
   while T.Link_Part /= Ptr loop
      T := T.Link_Part;
   end loop;
   -- T POINTS TO CELL BEFORE Ptr
   T.Link_Part := new List_Cell_Type'(X, Ptr);
end if;
```

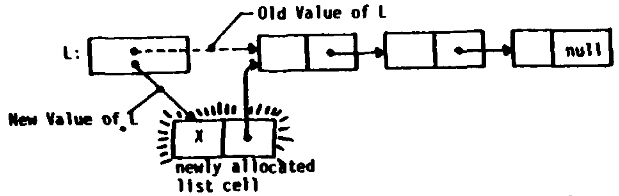
MAKE SURE THE CLASS UNDERSTANDS THE USE OF ALLOCATORS ON THE SLIDE.

THE DIAGRAMS CAN BE MADE MORE VIVID BY COPYING THEM ONTO A BLACKBOARD AND USING AN ERASER TO SHOW HOW THE DATA STRUCTURE CHANGES DURING EACH STEP OF THE INSERTION.

IMPLEMENTATION OF LIST OPERATIONS USING LINKED LISTS (Continued)

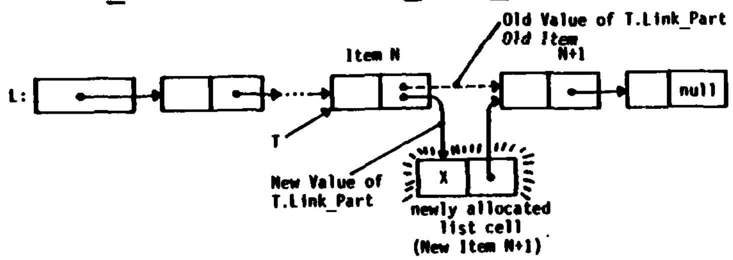
- **INSERT VALUE X AT FRONT OF LIST L:**

```
L := new List_Cell_Type'(X, L);
```



- **INSERT VALUE X AFTER POSITION N (N > 0):**

```
T := L;
for I in 1 .. N - 1 loop
   T := T.Link_Part;
end loop;
T.Link_Part := new List_Cell_Type'(X, T.Link_Part);
```



- **INSERT VALUE X AFTER POSITION POINTED TO BY Ptr:**

```
Ptr.Link_Part := new List_Cell_Type'(X, Ptr.Link_Part);
   -- DIAGRAM ABOVE WITH T REPLACED BY Ptr
```

- **INSERT VALUE X BEFORE POSITION N:**

SAME AS INSERTING AFTER POSITION N - 1 IF N > 1.
SAME AS INSERTING AT FRONT OF LIST OTHERWISE

- **INSERT VALUE X BEFORE POSITION Ptr:**

DIFFICULT

INSTRUCTOR NOTES

DELETION OF THE FIRST CELL MUST BE HANDLED SPECIALLY BECAUSE THE VARIABLE L IS UPDATED RATHER THAN THE Link_Part OF SOME LIST CELL.

DELETION OF THE LIST CELL POINTED TO BY P IS DIFFICULT FOR THE SAME REASON THAT INSERTION BEFORE THE LIST CELL POINTED TO BY P IS DIFFICULT -- WE NEED A WAY TO FIND THE PREVIOUS CELL OF THE LIST. AS BEFORE, THIS CAN BE ACCOMPLISHED BY STARTING AT THE BEGINNING OF THE LIST AND SEARCHING FOR THE PREDECESSOR OF THE CELL IN QUESTION.

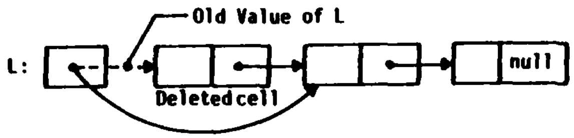
THE DELETED CELL STILL POINTS TO PART OF THE LIST. HOWEVER, IT IS NO LONGER <u>POINTED TO</u> FROM WITHIN THE LIST, SO IT IS NO LONGER CONSIDERED PART OF THE LIST. IN FACT, IF NO OTHER ACCESS OBJECTS POINT TO THE DELETED CELL, IT IS UNREACHABLE. FOR ALL INTENTS AND PURPOSES, IT NO LONGER EXISTS.

AS WITH THE PREVIOUS SLIDE, A DYNAMIC VERSION OF THE DIAGRAM, USING ERASER AND CHALK, MAY BE HELPFUL.

IMPLEMENTATION OF LIST OPERATIONS USING LINKED LISTS (Continued)

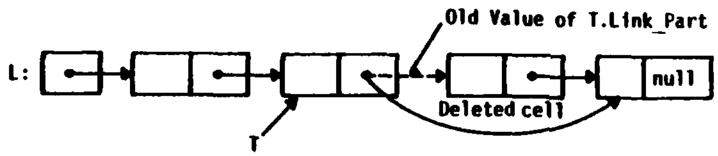
- DELETE VALUE AT FRONT OF NON-EMPTY LIST L:

```
L := L.Link_Part;
```



- DELETE VALUE AT POSITION N (N > 1):

```
T := L;
for I in 1 .. N - 2 loop
   T := T.Link_Part;
end loop;
T.Link_Part := T.Link_Part.Link_Part;
```



- DELETE VALUE AT POSITION Ptr:

DIFFICULT

- DELETE VALUE AFTER POSITION N:

SAME AS DELETING VALUE AT POSITION N + 1

- DELETE VALUE AFTER POSITION Ptr:

```
Ptr.Link_Part := Ptr.Link_Part.Link_Part;
   -- DIAGRAM ABOVE WITH T REPLACED BY Ptr
```

INSTRUCTOR NOTES

IN THE NEW TYPE DECLARATIONS, List_Cell_Pointer_Type PLAYS THE ROLE PREVIOUSLY PLAYED BY List_Type.

List_Type IS REDEFINED AS A RECORD CONSISTING OF TWO COMPONENTS -- THE LIST AS PREVIOUSLY ENVISIONED AND THE LENGTH OF THE LIST.

IMPLEMENTATION OF LIST OPERATIONS USING LINKED LISTS (Continued)

- CURRENT LENGTH OF L:

```
Current_Length := 0;
T := L;
while T /= null loop
   Current_Length := Current_Length + 1;
   T := T.Link_Part;
end loop;
```

IF THIS IS A FREQUENT OPERATION, IT MAY BE WORTHWHILE TO REDEFINE List_Type AS FOLLOWS:

```
type List_Cell_Type;

type List_Cell_Pointer_Type is access List_Cell_Type;

type List_Cell_Type is
   record
      Item_Part : Item_Type;
      Link_Part : List_Cell_Pointer_Type;
   end record;

type List_Type is
   record
      Length_Part   : Natural := 0;
      Contents_Part : List_Cell_Pointer_Type;
   end record;
```

INSERTION AND DELETION OPERATIONS MUST BE CHANGED ACCORDINGLY, TO INCREMENT AND DECREMENT THE Length_Part COMPONENT, AND TO REFER TO L.Contents_Part INSTEAD OF L.

INSTRUCTOR NOTES

THE Link_Part OF THE FIRST CELL NOW CONTAINS WHAT WE PREVIOUSLY THOUGHT OF AS THE VALUE OF THE LIST.

UNIFORMITY RESULTS FROM THE FACT THAT EACH LIST CELL CONTAINING A GENUINE LIST ITEM IS PRECEDED BY SOME LIST CELL POINTING TO IT.

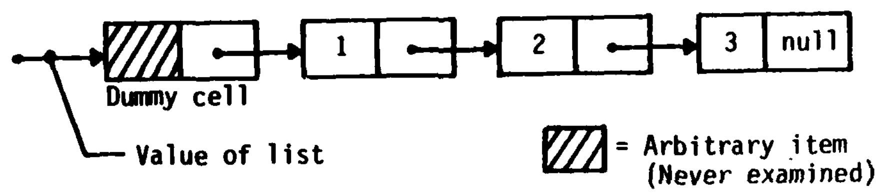
THE DUMMY CELL IS SOMETIMES CALLED A HEADER CELL.

IMPLEMENTATION OF THE LIST OPERATIONS USING THIS REPRESENTATION IS LEFT AS AN EXERCISE FOR THE CLASS.

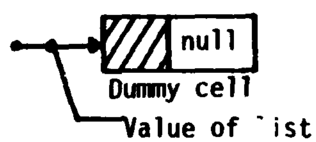
DUMMY LIST CELL

VARIATION: BEGIN EACH LIST WITH A "DUMMY" LIST CELL CONTAINING AN ARBITRARY ITEM.

-- LIST CONTAINING INTEGER 1, 2, 3



-- EMPTY LIST



BENEFIT: THE FIRST ITEM IN THE LIST AND SUBSEQUENT ITEMS ARE TREATED UNIFORMLY.

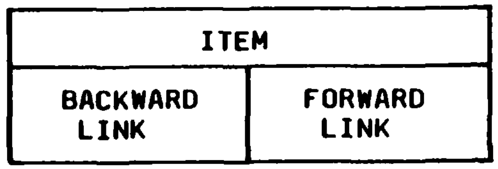
INSTRUCTOR NOTES

DOUBLY-LINKED LISTS ARE ESPECIALLY USEFUL WHEN THERE ARE POINTERS FROM OUTSIDE THE LIST POINTING DIRECTLY TO LIST CELLS, AND IT IS NECESSARY TO FIND THE PREDECESSOR OF A CELL POINTED TO FROM THE OUTSIDE.

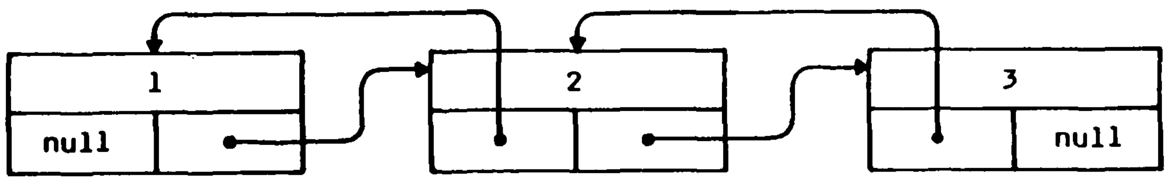
TYPE DECLARATIONS AND IMPLEMENTATION OF LIST OPERATIONS FOR THIS REPRESENTATION ARE LEFT AS AN EXERCISE FOR THE CLASS.

DOUBLY LINKED LIST

VARIATION: GIVE EACH LIST CELL A Forward_Link_Part POINTING TO THE NEXT CELL ON THE LIST AND A Backward_Link_Part POINTING TO THE PREVIOUS CELL ON THE LIST.



VALUE OF LIST



BENEFITS: LIST CAN BE TRAVERSED IN EITHER DIRECTION. NO NEED TO KEEP TRACK OF PREVIOUS LIST CELL WHEN DOING INSERTION OR DELETION.

INSTRUCTOR NOTES

THIS IS A COMBINATION OF THE PREVIOUS TWO VARIATIONS.

SHADING REPRESENTS AN ARBITRARY VALUE.

THE FORWARD LINKS FORM A CLOCKWISE RING AND THE BACKWARD LINKS FORM A COUNTERCLOCKWISE RING.

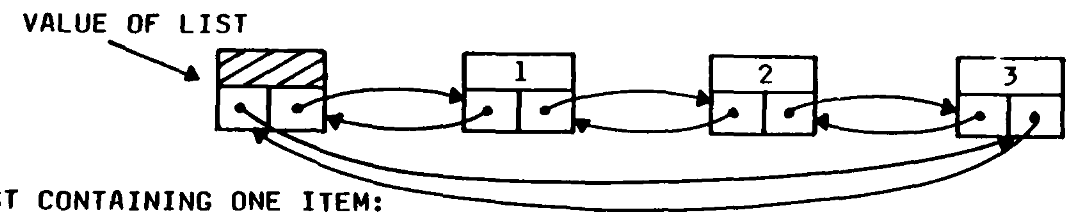
IMPLEMENTATION OF LIST OPERATIONS FOR THIS REPRESENTATION IS LEFT AS AN EXERCISE FOR THE CLASS.

DOUBLY LINKED LIST WITH DUMMY CELL

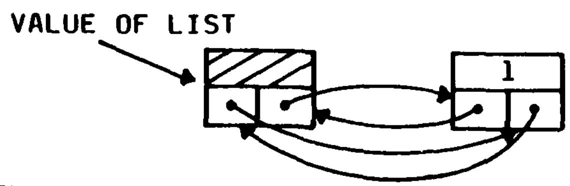
VARIATION: ADD A "DUMMY" LIST CELL CONTAINING AN ARBITRARY ITEM. THE CELL IS POSITIONED BOTH AS THE PREDECESSOR OF THE FIRST REGULAR CELL AND THE SUCCESSOR OF THE LAST REGULAR CELL.

THE "VALUE OF THE LIST" IS A POINTER TO THE DUMMY CELL.

-- LIST CONTAINING INTEGERS 1, 2, 3:



-- LIST CONTAINING ONE ITEM:



-- EMPTY LIST:

BENEFITS: UNIFORM TREATMENT OF FIRST, LAST, AND INTERMEDIATE CELLS
DIRECT ACCESS TO LAST CELL ON LIST (E.G., TO TRAVERSE ENTIRE LIST OR REVERSE ORDER)

INSTRUCTOR NOTES

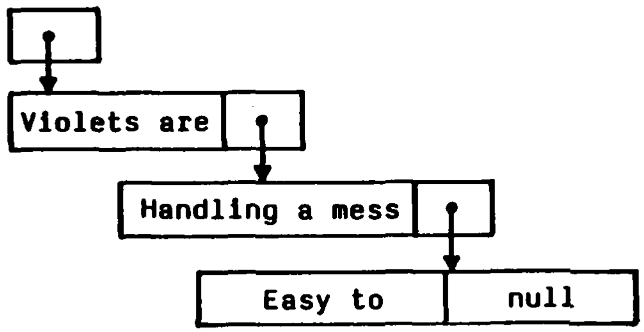
STACK SIZE MAY BE IMPLICITLY LIMITED BY THE AMOUNT OF STORAGE AVAILABLE FOR CREATING ALLOCATED VARIABLES.

LIST OPERATIONS DEALING ONLY WITH THE FIRST ITEM ON A LIST ARE VERY SIMPLE AND EFFICIENT. SINCE STACK OPERATIONS ONLY DEAL WITH THE TOP STACK ELEMENT, THIS IS A SIMPLE AND EFFICIENT REPRESENTATION.

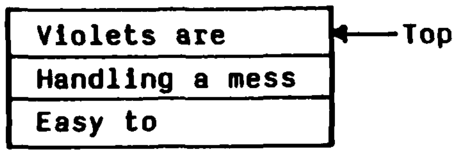
LINKED LIST REPRESENTATION OF STACKS

FIRST ITEM ON LIST REPRESENTS TOP OF STACK.

Partial_Message_Stack



CORRESPONDS TO:



TO PUSH: ADD A LIST CELL TO FRONT OF LIST

TO POP: EXAMINE AND REMOVE FIRST LIST CELL

THE EMPTY STACK IS REPRESENTED BY null.

NO EXPLICIT LIMIT ON MAXIMUM STACK SIZE.

INSTRUCTOR NOTES

IT MAY BE HELPFUL TO ILLUSTRATE THE PUSH AND POP OPERATIONS STEP-BY-STEP ON THE BLACKBOARD.

ANSWERS:

```
Partial Message Stack IS EMPTY
   Partial_Message_Stack = null

POP Partial_Message_Stack
   Current_Message := Partial_Message_Stack.Item_Part;
   Partial_Message_Stack := Partial_Message_Stack.Link_Part;
```

LINKED LIST REPRESENTATION OF STACKS (Continued)

```
type Stack_Cell_Type;

type Stack_Type is access Stack_Cell_Type;

type Stack_Cell_Type is
   record
      Item_Part : Message_Type;
      Link_Part : Stack_Type;
   end record;
```

<u>Partial Message Stack : Stack Type := (EMPTY STACK);</u>

```
      Partial_Message_Stack : Stack_Type := null;
```

<u>Partial Message Stack IS EMPTY</u>

<u>PUSH Current Message ONTO Partial Message Stack;</u>

```
      Partial_Message_Stack :=
         new Stack_Cell_Type'
            (Item_Part =>  Current_Message,
             Link_Part =>  Partial_Message_Stack);
```

<u>POP Partial Message Stack INTO Current Message;</u>

INSTRUCTOR NOTES

A QUEUE IS REPRESENTED AS A RECORD CONSISTING OF TWO COMPONENTS:

- A POINTER TO THE FIRST CELL ON THE LIST (A LIST VALUE IN THE SENSE WE HAVE USED IT PREVIOUSLY)

- A POINTER TO THE LAST CELL ON THE LIST

SINCE A QUEUE IS MANIPULATED ONLY AT ITS ENDS, PROVIDING DIRECT ACCESS TO THE FIRST AND LAST CELLS ON THE LIST ALLOWS FOR A SIMPLE AND EFFICIENT REPRESENTATION.

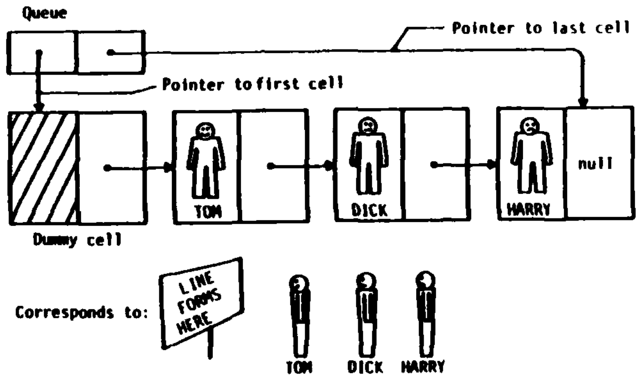
SINCE THERE IS A DUMMY CELL, EVERY LIST HAS A FIRST AND LAST CELL. IN A LIST WITHOUT ITEMS, THESE ARE BOTH THE DUMMY CELL.

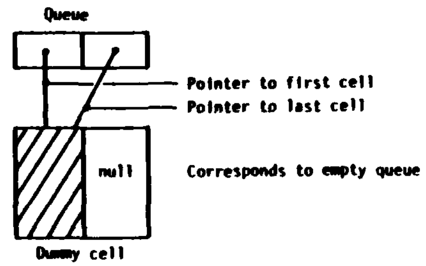
LIST LINKS POINT FROM THE FRONT OF THE QUEUE TO THE BACK. (THEY ANSWER THE QUESTION, "WHOSE TURN FOLLOWS MINE?")

QUEUE SIZE IS IMPLICITLY LIMITED BY THE AMOUNT OF STORAGE AVAILABLE FOR ALLOCATING CELLS.

LINKED LIST REPRESENTATION OF QUEUES

- FIRST ITEM ON LIST REPRESENTS FRONT OF QUEUE (OLDEST ITEM -- NEXT TO BE REMOVED)
- LAST ITEM ON LIST REPRESENTS BACK OF QUEUE (NEWEST ITEM -- MOST RECENTLY ADDED)
- BEGIN LIST WITH A DUMMY LIST CELL TO MAKE IT EASIER TO INSERT INTO AN EMPTY QUEUE.
- KEEP A POINTER TO THE LAST CELL ON THE LIST TO MAKE IT EASIER TO INSERT AT THE END OF THE LIST.





- NO EXPLICIT LIMIT ON QUEUE SIZE.

INSTRUCTOR NOTES

THESE ARE THE DATA DECLARATIONS FOR A QUEUE OF ORDERS, TO SOLVE THE INVENTORY MANAGEMENT PROBLEM GIVEN EARLIER.

LINKED LIST REPRESENTATION OF QUEUES (Continued)

```
type Queue_Cell_Type;

type Queue_Cell_Pointer_Type is access Queue_Cell_Type;

type Queue_Cell_Type is
   record
      Item_Part : Order_Type;
      Link_Part : Queue_Cell_Pointer_Type;
   end record;

type Queue_Type is
   record
      First_Cell_Part, Last_Cell_Part : Queue_Cell_Pointer_Type;
   end record;
```

INSTRUCTOR NOTES

THIS IS THE FIRST OF THREE SLIDES ILLUSTRATING THE IMPLEMENTATION OF QUEUE OPERATIONS FOR A LINKED QUEUE.

THERE ARE SEVERAL CONTEXTS IN WHICH ADA EVALUATES AN EXPRESSION ONCE FOR EACH TIME ITS VALUE IS USED:

```
A, B, C : T := E;    -- E EVALUATED THREE TIMES

procedure P (A, B, C : in T := E);
       -- E EVALUATED ONCE FOR EACH MISSING
       -- ACTUAL PARAMETER EACH TIME P IS CALLED

P (A | B | C =>  E);    -- E EVALUATED THREE TIMES WHENEVER
                        -- THE PROCEDURE CALL IS EXECUTED

type T1 is
   record
      A, B, C : T2 := E;
   end record
   -- E EVALUATED THREE TIMES WHENEVER THE DECLARATION
   -- OF A T1 OBJECT WITHOUT ITS OWN INITIAL VALUE
   -- IS ELABORATED.

T1'(A | B | C =  E)    -- E EVALUATED THREE TIMES WHENEVER THE
                       -- THE RECORD AGGREGATE IS EVALUATED

(1 .. 10 =>  E)        -- E EVALUATED TEN TIMES WHENEVER
                       -- THE ARRAY AGGREGATE IS EVALUATED
```

LINKED LIST REPRESENTATION OF QUEUES (Continued)

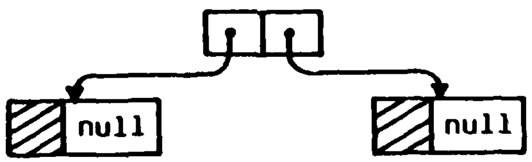
<u>Queue : Queue_Type := (empty queue);</u>

```
Arbitrary_Order : Order_Type;
Dummy_Cell_Pointer : Queue_Cell_Pointer_Type :=
                     new Queue_Cell_Type'
                          (Item_Part =>  Arbitrary_Order,
                           Link_Part =>  null);

Queue : Queue_Type :=
             (First_Cell_Part | Last_Cell_Part =>
                     Dummy_Cell_Pointer);
```

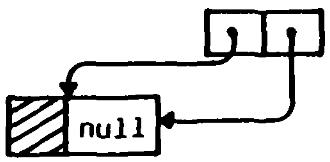
NOTE: THE FOLLOWING WOULD BE INCORRECT:

```
Queue : Queue_Type :=
             (First_Cell_Part | Last_Cell_Part =>
              new Queue_Cell_Type'(Arbitrary_Order, null));
```

THE ALLOCATOR WOULD BE EVALUATED ONCE FOR EACH Queue_Type RECORD COMPONENT, GIVING US



INSTEAD OF



Queue.First_Cell_Part ALWAYS POINTS TO THE DUMMY CELL. Queue.Last_Cell_Part POINTS TO THE DUMMY CELL IF AND ONLY IF THIS IS THE ONLY CELL ON THE LIST.

IT IS NOT MEANINGFUL TO TALK ABOUT A LINKED QUEUE BEING FULL. HOWEVER, AN ALGORITHM USING A QUEUE MAY CALL FOR CERTAIN ACTIONS TO BE PERFORMED ONCE A QUEUE HAS GROWN TO A CERTAIN SIZE. THE INVENTORY MANAGEMENT PROBLEM PRESENTED EARLIER CALLED FOR ORDERS ARRIVING WITH TEN CUSTOMERS WAITING TO BE IGNORED.

IF DETERMINING THE CURRENT LENGTH OF THE QUEUE IS A FREQUENT OPERATION, THE LENGTH CAN BE MAINTAINED AS A THIRD COMPONENT OF THE Queue_Type RECORD. IT WOULD BE INITIALIZED TO ZERO WHEN CONSTRUCTING AN EMPTY QUEUE, INCREMENTED WHEN INSERTING AN ITEM, AND DECREMENTED WHEN REMOVING AN ITEM. AN EMPTY QUEUE COULD THEN BE DETECTED SIMPLY BY COMPARING THIS COMPONENT WITH ZERO.

LINKED LIST REPRESENTATION OF QUEUES (Continued)

Queue IS EMPTY

```
Queue.Last_Cell_Part = Queue.First_Cell_Part
```

Queue HAS N ITEMS

```
Count := 0;
P := Queue.First_Cell_Part.Link_Part;
while P /= null loop
   Count := Count + 1;
   P := P.Link_Part;
end loop;

if Count = N ...
```

INSTRUCTOR NOTES

A BLACKBOARD DIAGRAM ILLUSTRATING INSERTION AND DELETION MAY BE HELPFUL.

REFER STUDENTS TO DIAGRAM ON 5-16.

FIRST ORDER:

```
Queue.First_Cell_Part.Link_Part.Item_Part
```

INSERT:

```
Queue.Last_Cell_Part.Link_Part :=
   new Queue_Cell_Type'(Item_Part =>  X, Link_Part =>  null);
Queue.Last_Cell_Part := Queue.Last_Cell_Part.Link_Part;
```

LINKED LIST REPRESENTATION OF QUEUES (Continued)

<u>FIRST ORDER IN Queue</u>

<u>INSERT ORDER X IN Queue</u>

<u>REMOVE FIRST ITEM FROM Queue</u> (ASSUMING QUEUE IS NOT EMPTY)

```
-- To handle deleting from list containing only 1 item.
-- Last_Cell_Part would point nowhere
if Queue.First_Cell_Part.Link_Part = Queue.Last_Cell_Part then
   Queue.Last_Cell_Part := Queue.First_Cell_Part;
end if;

Queue.First_Cell_Part.Link_Part :=
   Queue.First_Cell_Part.Link_Part.Link_Part;
```

INSTRUCTOR NOTES

THE SLIDES ON MULTILISTS DO NOT DEAL DIRECTLY WITH Ada.

ONLY A BROAD OVERVIEW IS GIVEN.

THE MAIN MESSAGE TO BE CONVEYED IS THAT THE NOTION OF LINKED LISTS IS VERSATILE, AND CAN BE GENERALIZED TO PRODUCE MORE POWERFUL DATA STRUCTURES.

IF SHORT ON TIME, THIS SECTION MAY BE GLOSSED OVER QUICKLY.

MULTILISTS

SOMETIMES THE DATA IN A LIST CELL BELONGS ON SEVERAL KINDS OF LISTS AT ONCE.

EXAMPLE: DATA ABOUT THE NEW YORK YANKEES MIGHT BELONG ON A LIST OF PROFESSIONAL BASEBALL TEAMS AND ON A LIST OF NEW YORK SPORTS TEAMS.

EACH LIST CELL CAN BE GIVEN A SEPARATE LINK FIELD FOR EACH KIND OF LIST IT MAY BELONG TO.

THIS KIND OF DATA STRUCTURE IS CALLED A MULTILIST.

INSTRUCTOR NOTES

THIS IS A MULTILIST WITH EACH CELL CONTAINING DATA ABOUT A SPORTS TEAM.

EACH CELL BELONGS TO TWO LISTS -- A LIST OF TEAMS IN A PARTICULAR CITY AND A LIST OF TEAMS PLAYING A PARTICULAR SPORT.

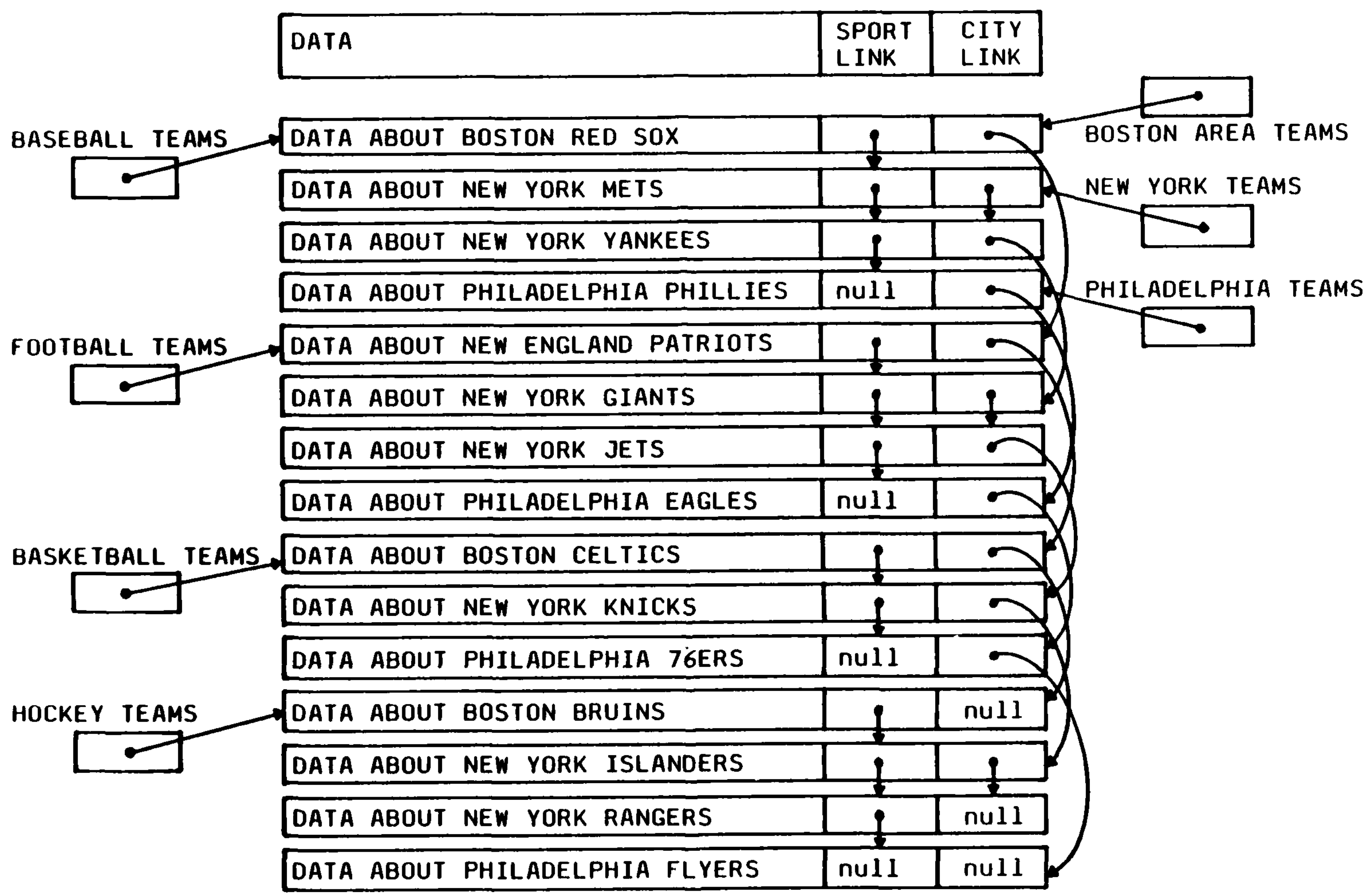
TO FIND ALL NEW YORK TEAMS, START WITH THE FIRST CELL POINTED TO BY THE <u>NEW YORK TEAMS</u> BOX AND FOLLOW THE CITY LINK POINTERS UNTIL <u>null</u> IS REACHED. (DEMONSTRATE THIS.)

TO FIND ALL BASEBALL TEAMS, START WITH THE FIRST CELL POINTED TO BY THE <u>BASEBALL TEAMS</u> BOX AND FOLLOW THE SPORT LINK POINTERS UNTIL <u>null</u> IS REACHED. (DEMONSTRATE THIS.)

NOTE THAT THE NEW YORK YANKEES AND THE NEW YORK METS CELLS BOTH APPEAR ON BOTH LISTS.

(<u>NOTE</u>: FOR THE SAKE OF CREATING AN EXAMPLE, IT WAS ASSUMED THAT THE SPORT PLAYED BY THE NEW YORK METS IS BASEBALL. THIS SEEMS TO BE A BETTER APPROXIMATION THAN ANY OF THE OTHER THREE SPORTS LISTED.)

MULTILISTS (Continued)



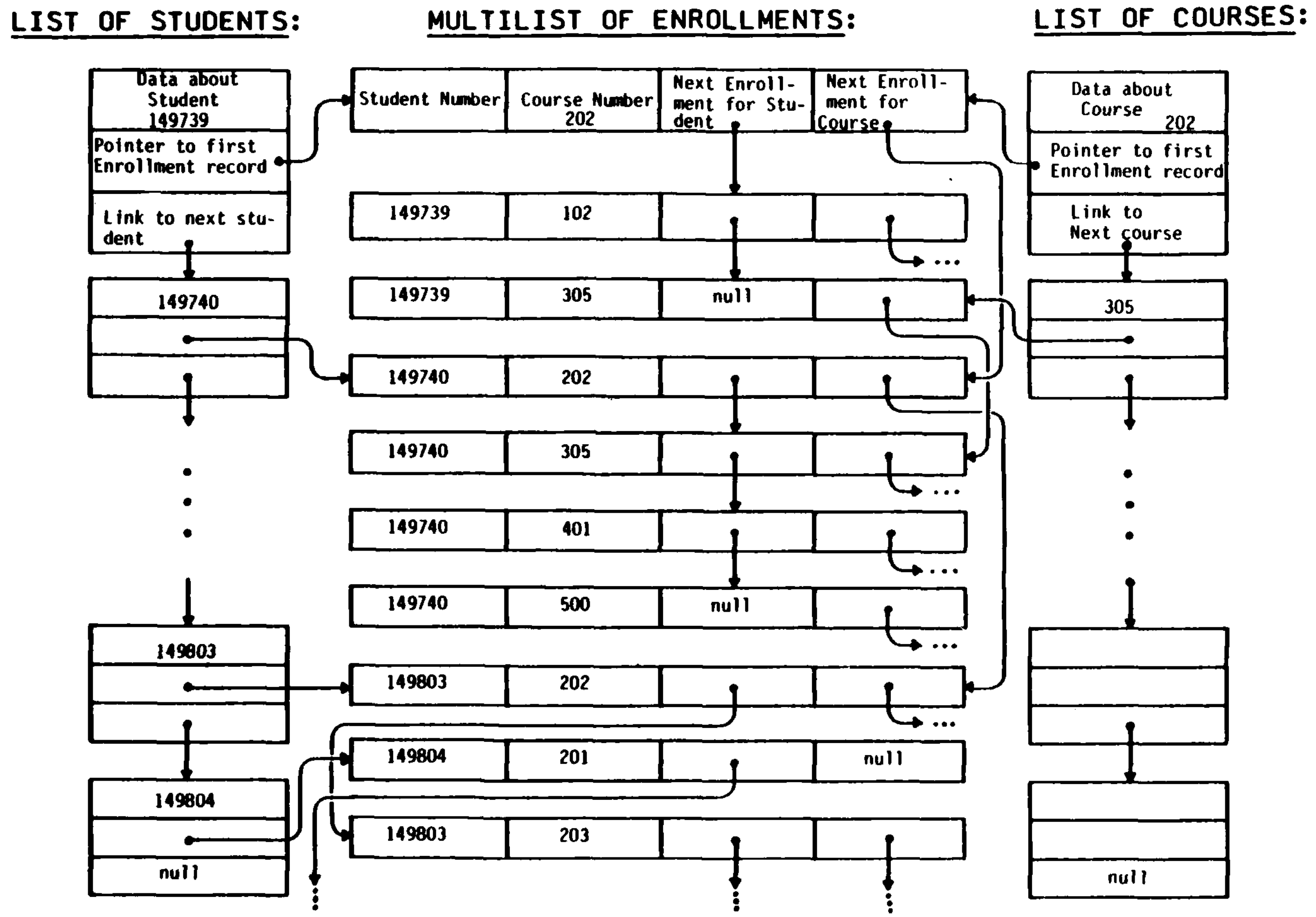
INSTRUCTOR NOTES

THE LISTS OF STUDENTS AND COURSES ARE ORDINARY LINKED LISTS. A CELL ON THIS LIST CONTAINS DATA ABOUT AN INDIVIDUAL STUDENT OR AN INDIVIDUAL COURSE. THE MULTILIST CONTAINS DATA RELATING STUDENTS TO COURSES.

SHOW THE LINKS THAT WOULD BE FOLLOWED TO ANSWER THE TWO QUERIES.

APPLICATION OF MULTILISTS: DATABASES

QUERY 1: LIST THE STUDENT NUMBERS OF ALL STUDENTS TAKING COURSE L305.
QUERY 2: LIST THE NUMBER OF EACH COURSE BEING TAKEN BY STUDENT 149740.



INSTRUCTOR NOTES

THE FIRST AND THIRD ENROLLMENT RECORDS ARE BOTH ON THE LIST FOR THE COURSE SHOWN, SO BOTH HAVE BACK-POINTERS TO THAT COURSE RECORD.

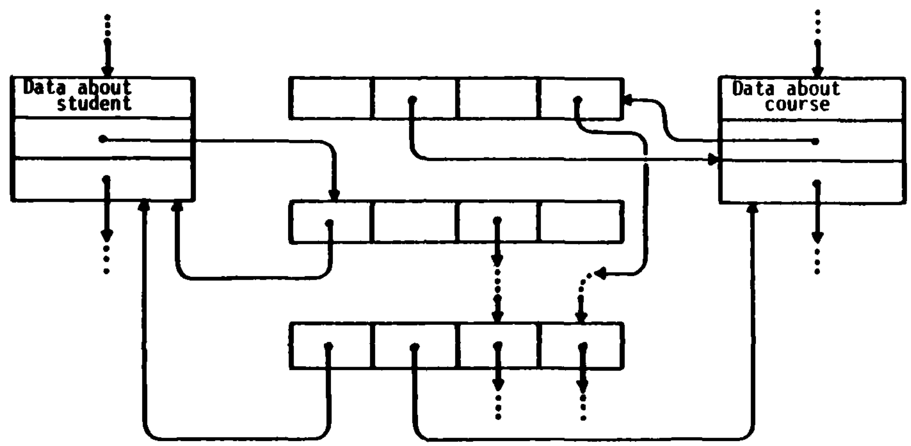
THE SECOND AND THIRD ENROLLMENT RECORDS ARE BOTH ON THE LIST FOR THE STUDENT SHOWN, SO BOTH HAVE BACK-POINTERS TO THAT STUDENT RECORD.

DATA IN A STUDENT RECORD MAY INCLUDE YEAR (FRESHMAN, SOPHOMORE, ETC.) AND GRADE POINT AVERAGE.

DATA IN A COURSE RECORD MAY INCLUDE DEPARTMENT.

APPLICATION OF MULTILISTS: DATABASES (Continued)

IN PRACTICE, ENROLLMENT RECORDS WOULD PROBABLY NOT CONTAIN STUDENT NUMBERS AND COURSE NUMBERS, BUT POINTERS BACK TO THE STUDENT RECORDS AND COURSE RECORDS ON WHOSE ENROLLMENT LISTS THEY RESIDE.



THIS ALLOWS ANSWERS TO QUERIES LIKE THE FOLLOWING:

- LIST THE NUMBER OF FRESHMEN IN EACH COURSE
- LIST THE GRADE POINT AVERAGES OF ALL STUDENTS SIMULTANEOUSLY ENROLLED IN TWO COMPUTER SCIENCE COURSES

THE BACK POINTERS ARE SOMETIMES CALLED <u>THREADS</u>.

LISTS CONTAINING THREADS ARE SOMETIMES CALLED <u>THREADED LISTS</u>.

INSTRUCTOR NOTES

ON AN ABSTRACT LEVEL, THE CONCEPT IS THE SAME. THE ONLY DIFFERENCE IS WHERE DATA IS STORED AND HOW WE PROVIDE ACCESS TO THAT DATA.

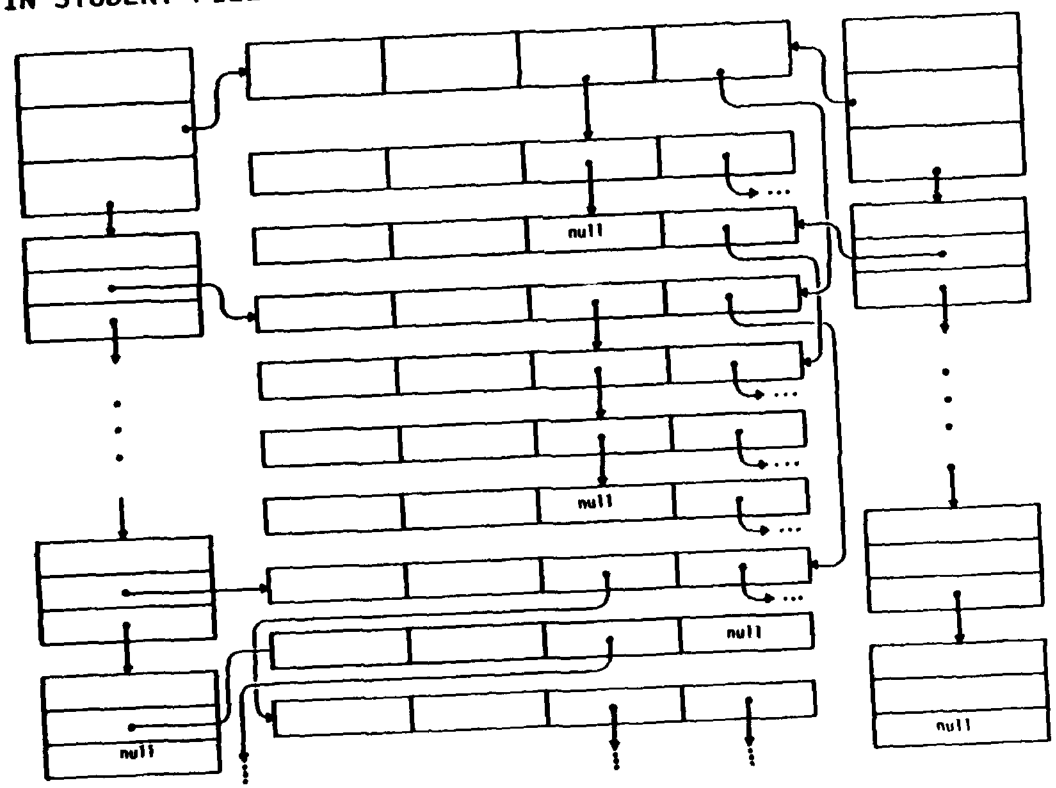
AN ARROW ON THIS SLIDE STANDS FOR THE KEY OF THE DIRECT ACCESS RECORD CORRESPONDING TO THE BOX THE ARROW POINTS TO. THE WORD <u>null</u> STANDS FOR A SPECIAL KEY NOT CORRESPONDING TO ANY RECORD.

APPLICATION OF MULTILISTS: DATABASES (Continued)

IN MOST DATABASE APPLICATIONS:

- BOXES ARE NOT ALLOCATED VARIABLES, BUT <u>ELEMENTS</u> (RECORDS) OF DIRECT ACCESS FILES.
- ARROWS ARE NOT ACCESS VALUES, BUT DIRECT ACCESS <u>KEYS</u>, EACH PROVIDING THE ABILITY TO READ OR WRITE A PARTICULAR RECORD OF THE FILE

RECORDS IN STUDENT FILE   RECORDS IN ENROLLMENT FILE   RECORDS IN COURSE FILE



VG 679.2

INSTRUCTOR NOTES

FOR LARGE, VERY SPARSE MATRICES, THERE WILL BE A CONSIDERABLE SAVINGS. IF THE MATRIX IS TOO SMALL OR TOO FULL OF NON-ZERO ELEMENTS, THE MULTILIST REPRESENTATION WILL REQUIRE MORE SPACE.

IN ANY EVENT, ACCESS IS SLOWER WITH A MULTILIST REPRESENTATION.

APPLICATION OF MULTILISTS: SPARSE MATRICES

A SPARSE MATRIX IS A MATRIX IN WHICH MOST OF THE MATRIX ELEMENTS ARE ZERO.

REPRESENTING SUCH A MATRIX AS A TWO-DIMENSIONAL ARRAY IS WASTEFUL OF SPACE, ESPECIALLY IF THE MATRIX IS LARGE.

SOLUTION: MULTILIST WITH ONE CELL FOR EACH NON-ZERO MATRIX ELEMENT.

EACH CELL RESIDES ON TWO LISTS:

-- THE LIST OF NON-ZERO ELEMENTS IN THE CORRESPONDING ELEMENT'S ROW

-- THE LIST OF NON-ZERO ELEMENTS IN THE CORRESPONDING ELEMENT'S COLUMN

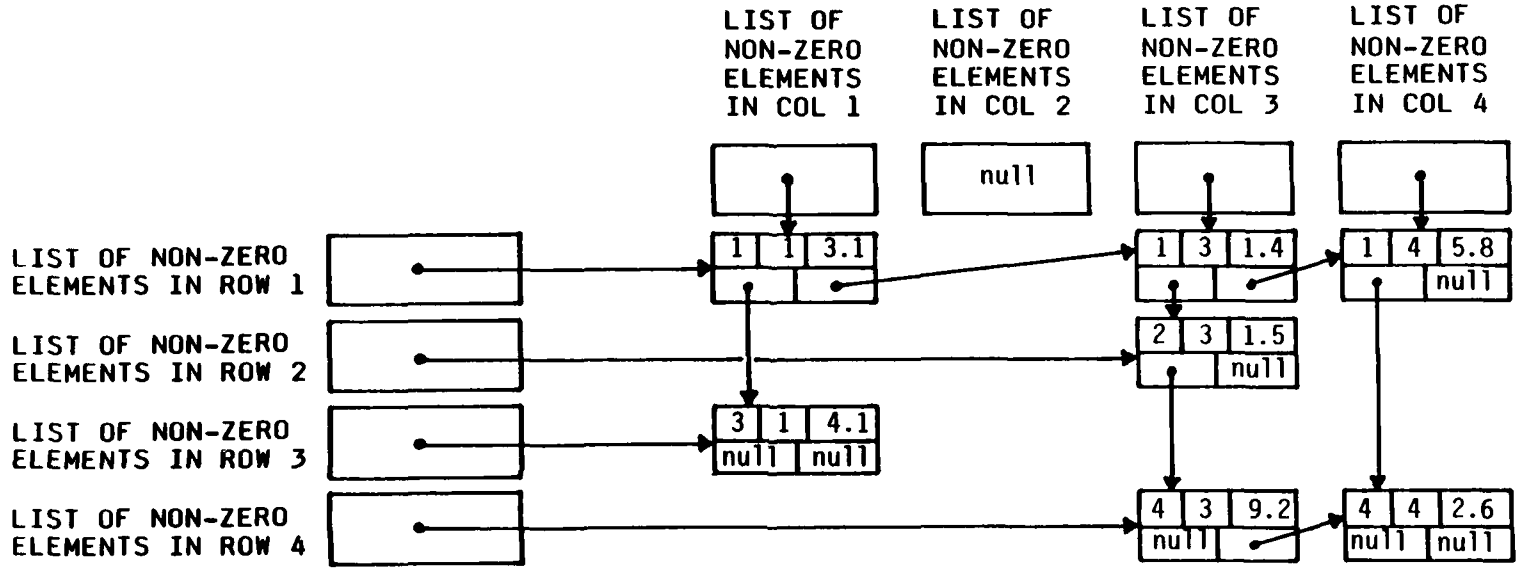
INSTRUCTOR NOTES

THIS REPRESENTATION FACILITATES TYPICAL OPERATIONS ON MATRICES, SUCH AS COMPUTING THE SUMS OF TWO ROWS.

DOUBLY LINKED LISTS FOR EACH ROW AND COLUMN ARE ALSO COMMON. THEN EACH ELEMENT CELL CONTAINS FOUR ACCESS VALUES -- ROW FORWARD AND BACKWARD LINKS AND COLUMN FORWARD AND BACKWARD LINKS.

THE BOXES TO THE LEFT OF AND ABOVE THE MATRIX, HOLDING LIST VALUES, MIGHT BE KEPT IN TWO ARRAYS, ONE FOR ROW ELEMENT LISTS AND ANOTHER FOR COLUMN ELEMENT LISTS.

APPLICATION OF MULTILISTS: SPARSE MATRICES (CONTINUED)



STRUCTURE OF ELEMENT CELL:

ROW NUMBER	COLUMN NUMBER	ELEMENT VALUE
POINTER TO NEXT NON-ZERO ELEMENT IN SAME COLUMN		POINTER TO NEXT NON-ZERO ELEMENT IN SAME ROW

CORRESPONDS TO: $\begin{pmatrix} 3.1 & 0.0 & 1.4 & 5.9 \\ 0.0 & 0.0 & 1.5 & 0.0 \\ 4.1 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 9.2 & 2.6 \end{pmatrix}$

INSTRUCTOR NOTES

PART III - DATA ABSTRACTION

SECTION 6	DATA TYPE ENCAPSULATION
SECTION 7	PRIVATE TYPES
SECTION 8	LIMITED PRIVATE TYPES
SECTION 9	USE OF EXCEPTIONS

INSTRUCTOR NOTES

SECTION 6

DATA TYPE ENCAPSULATION

INSTRUCTOR NOTES

A BRIEF OVERVIEW OF TOPICS TO FOLLOW

ABSTRACT DATA TYPES

- ABSTRACT DATA TYPES INVOLVE THE FOLLOWING THREE CONCEPTS:

  1. <u>ABSTRACTION</u> - EXTRACTION AND PRESENTATION OF THE ESSENTIAL PROPERTIES OF A DATA TYPE WHILE OMITTING THE INESSENTIAL DETAILS, RELATIVE TO A GIVEN LEVEL OF USE.

  2. <u>INFORMATION HIDING</u> - MAKING INACCESSIBLE ALL IMPLEMENTATION DETAILS OF A DATA TYPE THAT ARE NOT ESSENTIAL *PROPERTIES OF THE* ABSTRACTION.

  3. <u>ENCAPSULATION</u> - GROUPING TOGETHER THE VARIOUS DETAILS OF A DATA TYPE ABSTRACTION AND ITS IMPLEMENTATION.

- THESE CONCEPTS ARE SUPPORTED IN Ada BY THE FOLLOWING Ada FEATURES:
  - PRIVATE TYPES
  - LIMITED PRIVATE TYPES
  - PACKAGES WITH PRIVATE PARTS.

INSTRUCTOR NOTES

THE NEXT SLIDE PROVIDES EXAMPLES OF HOW WE CAN CATEGORIZE SOME OF THE SCALAR AND COMPOSITE TYPES WE HAVE SEEN.

A DATA TYPE CONSISTS OF THREE THINGS:

1. A SET OF <u>VALUES</u>.

2. A SET OF <u>OPERATIONS</u>.

3. A SET OF <u>RELATIONSHIPS</u> BETWEEN OPERATIONS.

INSTRUCTOR NOTES

INTEGER:
THE AXIOMS GIVEN DO NOT ACCOUNT FOR A POSSIBLE Numeric_Error WHEN EXPRESSION VALUES ARE NOT IN THE RANGE: Integer'First .. Integer'Last.

CHARACTER:
THE GIVEN AXIOMS DO NOT ACCOUNT FOR A POSSIBLE Constraint_Error AT THE END-POINTS OF THE Character RANGE.

STRING:
THE GIVEN AXIOMS ARE ACCURATE ONLY FOR NON-NULL STRINGS.
IN Ada PROGRAMS, ATTRIBUTES CANNOT REALLY BE APPLIED TO EXPRESSIONS LIKE (a & b).

- REVIEW: STRONGLY TYPED MEANS THAT OBJECTS OF A GIVEN TYPE MAY TAKE ON ONLY THOSE VALUES THAT ARE APPROPRIATE TO THE TYPE AND THE ONLY OPERATIONS THAT MAY BE APPLIED TO AN OBJECT ARE THOSE DEFINED FOR ITS TYPE.

- POINT OUT Ada GIVES US A MARGIN OF SAFETY BY DETECTING TYPE CONFLICTS AT COMPILATION TIME. THUS, WE PROBABLY GET MORE ERRORS DURING PROGRAM COMPILATION BUT WE HAVE GREATER CONFIDENCE THAT OUR PROGRAMS ARE CORRECT DURING EXECUTION.

Ada IS A STRONGLY TYPED LANGUAGE

	VALUES	OPERATIONS	RELATIONSHIPS
Integer	Integer'First, ..., -1, 0, 1, ..., Integer'Last	+, -, *, /, rem, mod, **, abs, = /= < > <= >=	A - B = A + (-B) A + 0 = A A + B = B + A A + (B + C) = (A + B) + C
Boolean	False, True	and or xor not and then or else = /= < > <= >= := conversion qualification membership ...	not (A and B) = (not A) or (not B) not (A or B) = (not A) and (not B) ...
Character	ASCII.nul, ..., '0', ..., '9', ..., 'A', ..., '2', ..., ASCII.del	= /= < > <= >= := conversion qualification membership	Character'Pos (Character'Succ(C)) = Character'Pos (C) + 1 Character'Val (Character'Pos (C)) = C ...
String	"", ..., "A", ..., "AA", "AB", ..., "AAA", "AAB", ...	& Indexing Slicing Aggregates = /= < > <= >= := conversion qualification membership attributes ...	S'Length = S'Last - S'First + 1

INSTRUCTOR NOTES

- WE WILL NOW CONSIDER THE VALUES, OPERATIONS, AND RELATIONSHIPS IN A USER DEFINED DATA TYPE.
- THIS IS THE VARIABLE-LENGTH LINEAR LIST EXAMPLE FROM PART II.
- THE AXIOMS ARE GIVEN INFORMALLY SINCE WE DON'T YET HAVE ANY SUBPROGRAM NAMES AND PARAMETER SPECIFICATIONS FOR THE OPERATIONS.
- THESE INFORMAL AXIOMS DON'T ACCOUNT FOR POSSIBLE EXCEPTIONS.
- DETAILS ABOUT Item_Type ARE IRRELEVANT HERE. JUST ASSUME IT HAS BEEN DECLARED EARLIER.

EXAMPLE: TYPE List_Type

```
type List_Storage_Space is array (1 .. Max_List_Length) of Item_Type;

type List_Type is
  record
       Current_Length : Natural range 0 .. Max_List_Length := 0;
       Elements       : List_Storage_Space;
  end record;
```

<u>VALUES</u>:

- (0, (others => junk)),
- (1, (1 =>Item, others => junk)), ...,
- (2, (1 =>Item1, 2 =>Item2, others => junk)), ...

<u>OPERATIONS</u>:

- LENGTH
- ACCESSING N-TH ELEMENT
- INSERTION
- DELETION
- AGGREGATES
- :=
- ETC.

<u>RELATIONSHIPS</u>:

- AXIOMS FOR LIST OPERATIONS, E.G.,
  - INSERTION INCREASES THE CURRENT LENGTH BY 1,
  - DELETION DECREASES THE CURRENT LENGTH BY 1,
  - INSERTING AN ITEM BEFORE THE N-TH ELEMENT MAKES IT THE NEW N-TH ELEMENT, AND THE OLD N-TH ELEMENT THE NEW (N+1)-TH ELEMENT ETC.,
  - DELETING THE N-TH ELEMENT MAKES THE (N+1)-TH ELEMENT THE NEW N-TH ELEMENT ETC.

INSTRUCTOR NOTES

THIS INLINE IMPLEMENTATION DOES NOT YET CHECK THE SPECIFIED CONSTRAINTS ON N AND List.Current_Length AND RAISE AN EXCEPTION WHEN THEY ARE VIOLATED.

NOTE THAT THE INSERT OPERATION REQUIRES THE LIST TO ALREADY HAVE AT LEAST ONE ELEMENT.

DO NOT GO OVER THE MECHANICS OF THE OPERATIONS.

POINT OUT THIS IS IN-LINE CODE. THE NEXT FEW SLIDES WILL EXPLAIN WHY THIS IS A POOR DESIGN CHOICE AND PROVIDE A BETTER ALTERNATIVE.

REVIEW OF LIST OPERATIONS

GIVEN:

```
List : List_Type;
```

CURRENT LENGTH OF List:

```
List.Current_Length
```

ACCESS THE N-TH ELEMENT OF List (WHERE 1 <= N <= List.CURRENT LENGTH):

```
List.Elements (N)
```

INSERT ITEM X BEFORE THE N-TH ELEMENT OF List

```
(WHERE 1 <= N <= List.Current_Length < Max_List_Length):

List.Elements (N + 1 .. List.Current_Length + 1) :=
   List.Elements (N .. List.Current_Length);
List.Elements (N) := X;
List.Current_Length := List.Current_Length + 1;
```

DELETE THE N-TH ELEMENT OF List (WHERE 1 <= N <= List.CURRENT LENGTH):

```
List.Elements (N .. List.Current_Length - 1) :=
   List.Elements (N + 1 .. List.Current_Length);
List.Current_Length := List.Current_Length - 1;
```

INSTRUCTOR NOTES

THE SUBPROGRAMS WILL BE SHOWN LATER WHEN THE PACKAGE BODY IS DISCUSSED.

THE NEXT SLIDE WILL SHOW THE CORRECT APPROACH, I.E., ENCAPSULATING BOTH DATA TYPE AND OPERATIONS IN A PACKAGE.

DISADVANTAGES OF USING IN-LINE CODE FOR OPERATIONS

1. EVEN WHEN THE INLINE CODE IS AS SHORT AS THAT FOR LIST INSERTION AND DELETION, WRITING IT CORRECTLY EACH TIME IS DIFFICULT.

2. INLINE CODE FOR LIST INSERTION AND DELETION REQUIRES MORE PROGRAM CODE SPACE THAN SUBPROGRAM CALLS WOULD.

3. LATER, WHEN YOU WANT TO CHANGE THE DEFINITION OF AN OPERATION, IT WILL BE QUITE DIFFICULT TO LOCATE *EACH INLINE APPLICATION OF THE OPERATION* AND CHANGE IT CORRECTLY.

THESE DISADVANTAGES CAN BE OVERCOME BY

- USING SUBPROGRAMS FOR THE OPERATIONS
- ENCAPSULATING THE DATA TYPE AND THESE OPERATIONS IN A PACKAGE

INSTRUCTOR NOTES

THE SEMANTICS OF THESE OPERATIONS ARE INFORMALLY IMPLIED BY THE COMMENTS. A MORE PRECISE DESCRIPTION OF THE SEMANTICS WOULD BE NEEDED TO ACTUALLY USE THESE OPERATIONS.

SUBPROGRAMS OF A TYPE

- A SUBPROGRAM IS AN OPERATION FOR A TYPE T IF THE SUBPROGRAM HAS A FORMAL PARAMETER OF TYPE T OR RETURNS A RESULT OF TYPE T.

- THE FORMAL PARAMETER AND RESULT TYPES DO NOT ALL HAVE TO BE THE SAME TYPE.

EXAMPLE: LENGTH OPERATION FOR TYPE NATURAL

```
function Length (List : List_Type) return Natural;
```

EXAMPLE: WRITE OPERATION FOR List_Type

```
procedure Write_Element (List : in out List_Type; N : in Positive;
                         Item : in Item_Type);
```

INSTRUCTOR NOTES

POINT OF SLIDE: THESE SUBPROGRAMS SHOULD BE THOUGHT OF AS OPERATIONS ON THE DATA TYPE.

ABSTRACT DATA TYPES IN Ada

- A PACKAGE PROVIDES THE MECHANISM FOR CREATING ABSTRACT DATA TYPES
- A PACKAGE DECLARATION SPECIFIES INTERFACE
  - HOW USER OF ABSTRACTION VIEWS THINGS
  - USES ABSTRACT VALUES RATHER THAN REPRESENTATION
  - CONTRACT BETWEEN USER OF ABSTRACTION AND IMPLEMENTER
- PACKAGE BODY PROVIDES IMPLEMENTATION OF ABSTRACTION
  - PROVIDES JUST ONE OF MANY POSSIBLE IMPLEMENTATIONS
  - DETAILS HIDDEN
  - FULFILLS CONTRACT

INSTRUCTOR NOTES

- BULLET 1: AN IMPORTANT PRECEPT FOR BOTH PROGRAMMING IN GENERAL AND REUSABILITY

- SEE SOFTECH'S Ada REUSABILITY GUIDELINES, APRIL 1985, SECTION 5 FOR FURTHER INFORMATION.

INSTRUCTOR MAY POINT OUT HOW THE OPERATIONS FOR List_Type WE JUST DISCUSSED FIT INTO THESE CATEGORIES.

ABSTRACTION PROVIDED BY PACKAGES

- SHOULD BE COMPLETE FOR A GIVEN LEVEL OF A DESIGN

- SHOULD INCLUDE THE FOLLOWING CLASSES OF OPERATIONS (I.E., SUBPROGRAMS)

  - CREATION
  - TERMINATION
  - CONVERSION
  - STATE INQUIRY
  - INPUT/OUTPUT REPRESENTATION
  - STATE CHANGE

INSTRUCTOR NOTES

HERE IS THE List_Package

POINT OUT THAT ALL THESE DECLARATIONS CONSTITUTE THE ABSTRACTION
THE USE OF EXCEPTIONS WILL BE DISCUSSED IN DEPTH IN SECTION 9.

RELYING ON THE PREDEFINED EXCEPTIONS TO DETECT UNUSUAL BUT ANTICIPATED SITUATION IS USUALLY BAD PRACTICE BECAUSE THEY DO NOT PROVIDE A GUARANTEE THAT THE EXCEPTION HAS ACTUALLY BEEN RAISED BECAUSE OF THE ANTICIPATED SITUATION.

THE PACKAGE BODY FOR List_Package IS GIVEN ON THE NEXT SLIDE.

EXAMPLE : List_Type PACKAGE DECLARATION

```
package List_Package is

     type Item_Type is ...;
     Max_List_Length : constant := ...;
     type List_Storage_Space is ...;
     type List_Type is ...;
     Out_of_Bounds : exception;
     function Length (...) return ...;
     function Element_Value (...) return ...;
     procedure Write_Element (...);
     procedure Insert (...);
     procedure Delete (...);

end List_Package;
```

INSTRUCTOR NOTES

THE CONCEPT OF IMPLEMENTATION OF A PACKAGE WAS REVIEWED IN SECTION 1.

NOTE THAT THE RULES OF Ada REQUIRE THAT THE SUBPROGRAM BODIES GO IN THE PACKAGE BODY AND NOT IN THE PACKAGE DECLARATION.

EXAMPLE : List_Type PACKAGE BODY

- TO COMPLETE THE ABSTRACTION, IMPLEMENT THE OPERATIONS

```
package body List_Package is
     function Length (...) return ... is
         ...
     begin
         ...
     end Length;

     function Element_Value (...) return
         ...
     is
         ...
     begin
         ...
     end Element_Value;

     procedure Write_Element (...) is
         ...
     begin
         ...
     end Write_Element;

     procedure Insert (...) is
         ...
     begin
         ...
     end Insert;

     procedure Delete (...) is
         ...
     begin
         ...
     end Delete;

end List_Package;
```

INSTRUCTOR NOTES

NOTE THAT A SUBPROGRAM DECLARATION PROVIDES ONLY A VERY LIMITED AMOUNT OF THE NECESSARY SEMANTIC INFORMATION FOR AN OPERATION, NAMELY, THE FORMAL PARAMETER AND RESULT SUBTYPES (WHICH OFTEN STILL ARE NOT SUFFICIENTLY CONSTRAINED, AS IN THE Element_Value EXAMPLE).

SOME RESEARCH LANGUAGES SUCH AS ALPHARD AND EUCLID HAVE SYNTACTIC MECHANISMS FOR SPECIFYING PRE- AND POST-CONDITIONS AND INVARIANTS, WHICH ARE CHECKED BY A VERIFIER AGAINST THE OPERATION BODIES.

THESE SPECIFICATIONS ARE MORE COMPLETE IN THAT THEY ACCURATELY DESCRIBE THE BOUNDARY CONDITIONS FOR WHEN EXCEPTIONS ARE TO BE RAISED.

THE FULL List_Type ABSTRACTION CONSISTS OF THE DECLARATIONS FOR Item_Type, Max_List_Length, List_Storage_Space, AND List_Type, AND THE DECLARATIONS AND SEMANTIC RELATIONSHIP COMMENTS FOR THE OPERATIONS Length, Element_Value, Write_Element, Insert, AND Delete.

SPECIFYING SEMANTIC RELATIONSHIPS

THE ONLY MECHANISM THAT Ada PROVIDES FOR SPECIFYING THESE RELATIONSHIPS IS VIA COMMENTS.

COMMENTS SHOULD BE PRECISE, COMPLETE, AND UNAMBIGUOUS. OTHERWISE, BOTH THE USER AND THE IMPLEMENTOR OF AN ABSTRACT DATA TYPE WILL BE UNCERTAIN AS TO EXACTLY WHAT THE ABSTRACTION IS.

EXAMPLE OF INCOMPLETENESS:

```
function Element_Value (List : List_Type; N : Positive) return Item_Type;
         -- Yield the N-th element of List.
```

COMPLETED EXAMPLE:

```
function Element_Value (List : List_Type; N : Positive) return Item_Type;
         -- YIELD THE N-th ELEMENT OF List IF N <= Length (List);
         -- OTHERWISE RAISE Out_of_Bounds EXCEPTION.
```

INSTRUCTOR NOTES

ONLY ONE EXAMPLE OF CODING IS SHOWN SINCE STUDENTS SHOULD NOT HAVE ANY DIFFICULTIES IN CODING THE SUBPROGRAMS WE HAVE TALKED ABOUT.

SUBPROGRAM IMPLEMENTATION

TO IMPLEMENT THE List_Type ABSTRACTION, ONE MUST CODE THE SUBPROGRAM BODIES FOR THE OPERATIONS.

EXAMPLE:

```
function Element_Value (List : List_Type; N : Positive) return Item_Type is
          -- YIELD THE N-th ELEMENT OF List IF N <= Length (List);
          -- OTHERWISE RAISE Out_of_Bounds.

begin -- Element_Value

  if N > Length (List) then
     raise Out_of_Bounds;
  end if;
  return List.Elements (N);

end Element_Value;
```

INSTRUCTOR NOTES

- THESE REASONS ARE DISCUSSED IN MORE DETAIL IN THE FOLLOWING SLIDES.
- THIS SECTION PREPARES THE CLASS FOR DISCUSSION OF PRIVATE TYPES IN THE NEXT SECTION.
- AGAIN LISTS ARE USED TO ILLUSTRATE THESE IDEAS.
- SO FAR, WE'VE ONLY ADDRESSED THE SEPARATION OF CONCERNS: DATA INTEGRITY WILL BE ADDRESSED THROUGH PRIVATE TYPES.

NEED FOR PRIVACY

- HIDE CERTAIN INFORMATION FROM THE USERS OF THE ABSTRACTIONS:

1. TO ENFORCE A SEPARATION OF CONCERNS ABOUT USE OF AN ABSTRACTION VERSUS IMPLEMENTATION OF AN ABSTRACTION.

2. FOR FLEXIBILITY IN CHOICE OF HOW AN ABSTRACTION IS IMPLEMENTED.

3. TO MAINTAIN THE INTEGRITY OF AN IMPLEMENTATION'S DATA BY RESTRICTING HOW IT IS MANIPULATED.

INSTRUCTOR NOTES

THIS SUBJECT WAS REVIEWED FOR PACKAGES IN SECTION 1.

BULLET 2 SUMMARIZES/JUSTIFIES THE MATERIAL JUST COVERED.

SEPARATION OF CONCERNS

- THE USE AND THE IMPLEMENTATION OF AN ABSTRACTION CAN BE STUDIED INDEPENDENTLY OF EACH OTHER.
- PACKAGES PROVIDE A MECHANISM FOR EXPRESSING THIS CONCERN.

INSTRUCTOR NOTES

- ONE NEED NEVER BE SIMULTANEOUSLY CONCERNED WITH BOTH THE USE OF AN ABSTRACTION (OR PACKAGE) IN OTHER PARTS OF A PROGRAM AND WITH THE IMPLEMENTATION OF THE ABSTRACTION.

FLEXIBILITY IN CHOICE OF IMPLEMENTATION

- MANY ABSTRACTIONS HAVE MORE THAN ONE REASONABLE IMPLEMENTATION.
- WHICH IS MOST REASONABLE INVOLVES SPACE/TIME TRADEOFFS AND DEPENDS ON WHICH OPERATIONS OF AN ABSTRACTION ARE MOST FREQUENTLY USED.
- IN ORDER TO CHOOSE BETWEEN IMPLEMENTATIONS OR TO SUBSEQUENTLY CHANGE AN IMPLEMENTATION, THE DETAILS OF THE IMPLEMENTATION MUST BE KEPT HIDDEN FROM THE USERS OF THE ABSTRACTION. OTHERWISE USERS OF THE PACKAGE CAN WRITE CODE DEPENDENT UPON A PARTICULAR IMPLEMENTATION.
- Ada <u>PACKAGE BODIES</u> PROVIDE A MECHANISM FOR HIDING IMPLEMENTATION DETAILS.

INSTRUCTOR NOTES

TECHNICALLY, A LINEAR LIST AND A LINKED LIST IMPLEMENT SLIGHTLY DIFFERENT ABSTRACTIONS UNLESS AN ARBITRARY UPPER BOUND IS IMPOSED ON THE LENGTH OF THE LINKED LIST. IGNORE THIS POINT UNLESS A STUDENT BRINGS IT UP.

AS WE HAVE ALREADY SEEN, ACCESSING THE N-TH ELEMENT OF A LIST IS FAST WITH A LINEAR REPRESENTATION AND SLOW WITH A LINKED REPRESENTATION, WHEREAS INSERTION AND DELETION ARE FAST WITH A LINKED REPRESENTATION AND SLOW WITH A LINEAR REPRESENTATION. THERE IS NO REPRESENTATION CHOICE THAT IS GOOD IN ALL SITUATIONS.

LIST EXAMPLE

NOTABLE POINTS

- IN THE ABSTRACT SENSE, A LIST IS JUST A SEQUENCE OF ELEMENTS.
- ITS REPRESENTATION AS A LINEAR OR LINKED LIST IS AN IMPLEMENTATION DETAIL.
- A USER SHOULD SEE ONLY THE ABSTRACT VALUE (SEQUENCE OF ELEMENTS).
- THE ACTUAL REPRESENTATION (LINKED OR LINEAR LIST) SHOULD REMAIN HIDDEN OR PRIVATE.

INSTRUCTOR NOTES

MAINTAINING THE INTEGRITY OF AN IMPLEMENTATION'S DATA

- IF THE INTERNAL DATA OF AN IMPLEMENTATION ARE AVAILABLE TO USERS THEN THEY WILL BE ABLE TO ACCIDENTALLY OR INTENTIONALLY ALTER IT SUCH THAT SOME OF THE INVARIANT PROPERTIES OR RELATIONSHIPS OF THE ABSTRACTION ARE VIOLATED.

- AN IMPLEMENTATION CANNOT MAINTAIN THE INTEGRITY OF ITS INTERNAL DATA UNLESS THAT DATA IS HIDDEN FROM THE USER OF THE ABSTRACTION.

- Ada'S <u>PACKAGE BODIES</u> PROVIDE AN APPROPRIATE MECHANISM FOR HIDING IMPLEMENTATION DATA.

EXAMPLE: LISTS

- WAYS OF CORRUPTING A LIST:

  1. ASSIGN DIRECTLY TO THE Current_Length COMPONENT OF A LINEAR LIST.

  2. ASSIGN DIRECTLY TO A LINKING COMPONENT OF A LINKED LIST (USING IN-LINE CODE).

INSTRUCTOR NOTES

MAKE SURE STUDENTS DO NOT CONFUSE INFORMATION HIDING OR PRIVACY WITH PHYSICAL SECRECY

- FEW STUDENTS WILL ACTUALLY BE CONFUSED ABOUT THIS

- REACHING THOSE STUDENTS WHO MIGHT BE, WILL SAVE YOU FROM ANSWERING SEEMINGLY STRANGE QUESTIONS COMING OUT OF NOWHERE

INFORMATION HIDING OR PRIVACY VERSUS PHYSICAL SECRECY.

- THE INFORMATION IN A PACKAGE BODY IS NOT KEPT PHYSICALLY SECRET FROM USERS OF THE PACKAGE. IT IS NOT RESTRICTED TO THOSE WITH A "NEED TO KNOW".

- THE USER OF A PACKAGE IS FREE TO READ THE SOURCE CODE OF A PACKAGE BODY. AND CERTAINLY THE Ada COMPILER MUST BE ABLE TO READ THE PACKAGE BODY.

- THE PURPOSE OF INFORMATION HIDING OR PRIVACY IS NOT TO PREVENT THE USER FROM KNOWING HOW A PACKAGE IS IMPLEMENTED, BUT TO PREVENT HIM FROM EXPLOITING THAT KNOWLEDGE SO AS TO VIOLATE:

  1. THE SEPARATION OF USAGE AND IMPLEMENTATION CONCERNS,

  2. THE FLEXIBILITY OF CHOICE OF IMPLEMENTATION,

  3. THE INTEGRITY OF IMPLEMENTATION DATA.

INSTRUCTOR NOTES

SECTION 7

PRIVATE TYPES

INSTRUCTOR NOTES

POINT OUT THE VISIBILITY OF THE Current_Length AND Elements COMPONENTS OF TYPE List_Type.

THE NEXT SLIDES SHOW SOME CONSEQUENCES OF THIS VISIBILITY.

PRIVATE TYPES

IN OUR List_Type EXAMPLE, ONLY THE IMPLEMENTATION DETAILS OF THE OPERATIONS HAVE BEEN HIDDEN (IN SUBPROGRAM BODIES) FROM USERS OF THE TYPE.

THE DETAILS OF HOW THE TYPE List_Type IS CONSTRUCTED FROM OTHER TYPES ARE STILL VISIBLE TO USERS OF THE TYPE.

```
package List_Package is

   type Item_Type is ...;
   Max_List_Length : constant := ...;

   type List_Storage_Space is array (1 .. Max_List_Length) of Item_Type;

   type List_Type is
      record
         Current_Length : Natural range 0 .. Max_List_Length := 0;
         Elements       : List_Storage_Space;
      end record;

   function Length (...) return ...;
   ...
   procedure Delete (...);

end List_Package;
```

INSTRUCTOR NOTES

- EXAMPLE 1 IS A RESULT OF List_Type BEING VISIBLY DECLARED AS A RECORD TYPE.

- WALK THROUGH EXAMPLE 2 SINCE IT MAY NOT BE INHERENTLY OBVIOUS THAT "+3 " APPENDS 3 JUNK ELEMENTS

PROBLEMS WITH VISIBLE DECLARATION OF List_Type

PROBLEM: USER'S HAVE DIRECT MANIPULATION OF THE REPRESENTATION.

EXAMPLES:

1. USERS CAN ACCESS THE Current_Length COMPONENT OF A LIST OBJECT TO OBTAIN THE LENGTH OF A LIST, INSTEAD OF USING THE LENGTH FUNCTION THAT THE ABSTRACTION HAS PROVIDED FOR THIS PURPOSE.

```
List_Object : List_Type := ...;
L1 : Natural := List_Object.Current_Length;  -- USES INFORMATION ABOUT HOW
                                             -- type List_Type IS COMPOSED.
L2 : Natural := Length (List_Object);        -- DOESN'T USE DETAILS OF
                                             -- List_Type COMPOSITION.
```

2. USERS COULD ASSIGN DIRECTLY TO THE Current_Length COMPONENT OF A LIST OBJECT AND CAUSE EITHER SOME TRAILING ELEMENTS TO BE DELETED OR SOME GARBAGE TO BE APPENDED (WHICH VIOLATES THE ABSTRACTION).

```
List_Object.Current_Length := Length (List_Object) - 3;  -- DELETES LAST 3
                                                         -- ELEMENTS.
List_Object.Current_Length := Length (List_Object) + 3;  -- APPENDS 3 JUNK
                                                         -- ELEMENTS.
```

INSTRUCTOR NOTES

PROBLEMS (Continued)

3. USERS CAN ACCESS INDIVIDUAL LIST ELEMENTS WITHOUT USING THE ABSTRACTION PROVIDED OPERATIONS Element_Value AND Write_Element.

```
List_Object : List_Type := ....;
Item : Item_Type := ...;
...
Item := List_Object.Elements (3);        -- USES DEFINITION OF List_Type.
Item := Element_Value (List_Object, 3); -- USES ABSTRACTION.
List_Object.Elements (3) := Item;        -- USES DEFINITION OF List_Type.
Write_Element (List_Object, 3, Item);   -- USES ABSTRACTION.
```

CONCLUSION:

- USING THE DEFINITION OF List_Type IN THESE WAYS DECREASES THE MAINTAINABILITY OF THE USER'S CODE. IF THE IMPLEMENTATION OF List_Type WERE CHANGED, E.G., TO A LINKED LIST REPRESENTATION, THEN THE USER'S CODE WOULD ALSO HAVE TO BE CHANGED.

INSTRUCTOR NOTES

POINT OUT THAT THE ONLY INFORMATION USERS KNOW ABOUT List_Type IS THAT THEY DON'T KNOW ITS REPRESENTATION. THIS REPRESENTATION INFORMATION IS FOR USE INSIDE THE PACKAGE ONLY.

THE PRIVATE PART WILL BE GIVEN IN A SUBSEQUENT SLIDE. HERE WE ARE ONLY INTERESTED IN THE VISIBLE ABSTRACTION.

THE INSTRUCTOR SHOULD GIVE AN EXAMPLE OF THE SECOND BULLET, I.E., SINCE THE USERS OF THE ABSTRACTION ARE PREVENTED FROM MANIPULATING THE REPRESENTATION DIRECTLY. IT DOESN'T MAKE SENSE FOR THEM TO USE PREDEFINED ADDITION, LOGIC OPERATIONS, ETC.

PRIVATE TYPES

PRIVATE TYPES PROVIDE A MECHANISM FOR HIDING THE COMPOSITION DETAILS OF AN ABSTRACT TYPE FROM THE USERS OF THE TYPE.

```
package List_Package is
   type Item_Type is ...;

   type List_Type is private;

   function Length (...) return ...;
   ...
end List_Package;
```

- THE REPRESENTATION (ARRAY, LINKED LIST) OF List_Type IS NOT VISIBLE TO A USER.
- PRIVATE TYPES RESTRICT WHICH PREDEFINED OPERATIONS CAN BE USED ON THE TYPE.
- AN ELEMENT OF THE LIST CANNOT BE ACCESSED OTHER THAN BY THE OPERATIONS PROVIDED BY THE PACKAGE.

INSTRUCTOR NOTE

POINT OUT THAT THE VIOLATIONS OF THE List_Type ABSTRACTION SHOWN ON THE PREVIOUS SLIDES ARE NO LONGER POSSIBLE NOW THAT List_Type IS A PRIVATE TYPE.

BENEFITS OF PRIVATE TYPES

- BY RESTRICTING WHAT THE USER OF THE ABSTRACTION CAN DO, WE'VE SOLVED THE PROBLEMS DISCUSSED AT THE BEGINNING OF THE SECTION.

  EXAMPLES:

  1. NOW THAT USERS CANNOT ACCESS A LENGTH COMPONENT, THEY MUST CALL THE Length FUNCTION PROVIDED IN THE PACKAGE SPECIFICATION.

  2. USERS CANNOT INTENTIONALLY DELETE GOOD ELEMENTS FROM OR APPEND GARBAGE ELEMENTS TO THE LIST; THEY MUST USE THE Insert AND Delete PROCEDURES TO CHANGE THE CONTENTS OF THE LIST.

  3. USERS MUST CALL Element_Value OR Write_Element TO ACCESS INDIVIDUAL LIST ELEMENTS BECAUSE THE LINKED LIST OR ARRAY REPRESENTATION IS NO LONGER VISIBLE.

INSTRUCTOR NOTE

LEAD IN FOR NEXT SLIDE:

NOTE IN PARTICULAR THAT AGGREGATES ARE NOT AVAILABLE AS AN OPERATION FOR CONSTRUCTING VALUES OF A PRIVATE TYPE, EVEN THOUGH THE PRIVATE TYPE MAY BE IMPLEMENTED AS AN ARRAY OR RECORD TYPE.

RECALL THAT THE INSERT OPERATION REQUIRES THAT THE LIST HAVE AT LEAST ONE ELEMENT IN ORDER TO INSERT AN ITEM BEFORE IT.

SOME EXAMPLES OF DISALLOWED PREDEFINED OPERATIONS ARE COMPONENT SELECTION, DEREFERENCING, ARITHMETIC, INDEXING, SLICES, ETC.

OPERATIONS

OPERATIONS AVAILABLE TO A USER OF List_Type:

1. PREDEFINED OPERATIONS
   - ASSIGNMENT (:=)
   - EQUALITY (=)
   - INEQUALITY (/=)
   - QUALIFIED EXPRESSIONS

2. THOSE EXPLICITLY DECLARED AS SUBPROGRAMS IN THE PACKAGE SPECIFICATION, FOR EXAMPLE, Length.

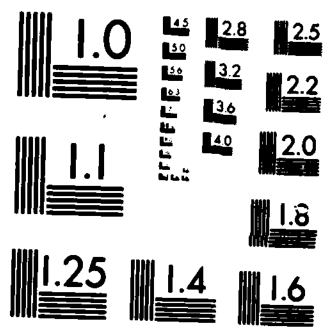
PREDEFINED OPERATIONS <u>NOT</u> ALLOWED:

1. AGGREGATES

2. ANY THAT DEPEND ON THE REPRESENTATION

AD-A165 075 ADA (TRADEMARK) TRAINING CURRICULUM: ADVANCED ADA 4/5
TOPICS L305 TEACHER'S GUIDE VOLUME 1(U) SOFTECH INC
WALTHAM MA 1986 DAAB07-83-C-K506

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

INSTRUCTOR NOTES

PREVIOUSLY, WE WERE USING AGGREGATES TO CONSTRUCT INITIAL List_Type VALUES.

FUNCTION Array_To_List IS THE CONSTRUCTION OPERATION FOR TYPE List_Type. SUBPROGRAMS Element_Value AND Write_Element ARE THE COMPONENT SELECTION OPERATIONS FOR TYPE List_Type.

SINCE AGGREGATES ARE NOT AVAILABLE FOR PRIVATE TYPES, IT IS CONVENIENT TO HAVE THE FOLLOWING CONVERSION OPERATIONS AS PART OF THE List_Type ABSTRACTION:

```
type Item_Array_Type is array (Positive range <>) of Item_Type;

function Array_To_List (Item_Array : Item_Array_Type) return List_Type;
                    -- YIELD A LIST OF THE ELEMENTS IN Item_Array.

function List_To_Array (List : List_Type) return Item_Array_Type;
                    -- YIELD AN ARRAY OF THE ELEMENTS IN List.
```

THE FUNCTION Array_To_List IS ESPECIALLY USEFUL AS A CONSTRUCTION OPERATION FOR CONSTRUCTING List_Type VALUES FROM A SEQUENCE OF Item_Type VALUES WHEN INITIALIZING List_Type VARIABLES OR CONSTANTS.

```
List_N : List_Type := Array_To_List ((Item_1, Item_2, ..., Item_N));
List_1 : List_Type := Array_To_List ((1 => Item_1));
List_0 : List_Type := Array_To_List ((1 .. 0 => Item_0));
```

INSTRUCTOR NOTES

PLUG IN THE APPROPRIATE DECLARATIONS AND COMMENTS AS GIVEN IN PREVIOUS SLIDES.

USER VISIBLE PART OF List_Type ABSTRACTION

```
package List_Package is

   type Item_Type is ...;
   type Item_Array_Type is ...;
   type List_Type is private;
   Out_Of_Bounds : exception;
   function Length (...) return ...;
   function Array_To_List (...) return ...;
   function List_To_Array (...) return ...;
   function Element_Value (...) return ...;
   procedure Write_Element (...);
   procedure Insert (...);
   procedure Delete (...);

end List_Package;
```

INSTRUCTOR NOTES

PACKAGES WITH PRIVATE PARTS

- ABSTRACT TYPE List_Type IS VISIBLE ONLY AS A PRIVATE TYPE; ITS COMPOSITION IS NOT VISIBLE TO A USER OF List_Type.

- HOWEVER, IN ORDER TO ACTUALLY BE ABLE TO USE List_Type IN OTHER Ada PROGRAM UNITS, AN Ada COMPILER MUST KNOW HOW TYPE List_Type IS COMPOSED OF OTHER TYPES.

- THUS, THE IMPLEMENTATION OF THE List_Type ABSTRACTION MUST SOMEWHERE GIVE A FULL TYPE DECLARATION FOR List_Type THAT DEFINES List_Type IN TERMS OF OTHER TYPES. FOR EXAMPLE:

```
Max_List_Length : constant := ...;

subtype List_Storage_Space is Item_Array_Type (1 .. Max_List_Length);

type List_Type is
   record
      Current_Length : Natural range 0 .. Max_List_Length := 0;
      Elements       : List_Storage_Space;
   end record;
```

INSTRUCTOR NOTES

REMIND STUDENTS THAT THE DECLARATION AND THE BODY OF List_Package CAN BE SEPARATELY COMPILED (E.G., AS LIBRARY UNITS). ONLY THE PACKAGE DECLARATION NEEDS TO BE COMPILED IN ORDER TO COMPILE OTHER UNITS THAT USE THE PACKAGE (E.G., VIA A <u>with</u> CLAUSE: with List_Package;).

SEPARATE COMPILATION CONCERNS

THE LOGICAL PLACE TO PUT THE FULL TYPE DECLARATION FOR List_Type IS IN THE BODY OF List_Package, ALONG WITH THE BODIES OF THE OPERATIONS FOR List_Type. THEN THE FULL TYPE DECLARATION WOULD BE HIDDEN FROM THE USERS OF List_Type.

SUPPOSE WE HAVE COMPILED THE DECLARATION, BUT NOT THE BODY, OF List_Package, AND WE NOW TRY TO COMPILE THE FOLLOWING:

```
with List_Package; use List_Package;
procedure P is
   List : List_Type := ...;
begin
   ... Length (List) ...
end P;
```

HOW DOES THE Ada COMPILER KNOW HOW MUCH SPACE TO ALLOCATE FOR VARIABLE List?

HOW DOES THE COMPILER KNOW HOW TO COMPILE INVOCATIONS OF List_Type OPERATIONS?

INSTRUCTOR NOTES

NOTE THAT THE DECLARATIONS IN THE PRIVATE PART ARE NOT PHYSICALLY SECRET FROM A USER OF THE PACKAGE. HOWEVER, A USER CANNOT EXPLOIT KNOWLEDGE OF THE PRIVATE PART; ONLY THE COMPILER CAN EXPLOIT SUCH KNOWLEDGE (AND MUST DO SO FOR PRIVATE TYPES).

NOTE THAT IF THE PRIVATE PART IS SUBSEQUENTLY CHANGED, THE RECOMPILATION RULES REQUIRE THAT ALL USES OF THE PACKAGE BE RECOMPILED, EVEN THOUGH THE PRIVATE PART IS "HIDDEN" FROM THESE USES.

(A PRIVATE PART IS ONLY REQUIRED WHEN THE VISIBLE PART DECLARES PRIVATE TYPES; OTHERWISE IT IS OPTIONAL. IT MAY ALSO CONTAIN ADDITIONAL HIDDEN DECLARATIONS.)

VISIBLE PART AND PRIVATE PART

IN ORDER FOR AN Ada COMPILER TO COMPILE USES OF A PRIVATE TYPE, IT MUST KNOW THE FULL TYPE DECLARATION THAT DEFINES HOW THE PRIVATE TYPE IS COMPOSED.

CONSEQUENTLY, IN Ada, A PACKAGE DECLARATION CONSISTS OF TWO PARTS:

1. <u>VISIBLE PART</u> CONTAINS DECLARATIONS THAT ARE VISIBLE TO USERS OF THE PACKAGE.

2. <u>PRIVATE PART</u> CONTAINS DECLARATIONS THAT ARE HIDDEN FROM USERS OF THE PACKAGE, BUT THAT ARE NEEDED BY THE COMPILER IN ORDER TO COMPILE USES OF THE VISIBLE DECLARATIONS. THE PRIVATE PART DECLARATIONS ARE VISIBLE ONLY WITHIN THE PRIVATE PART AND THE PACKAGE BODY.

A PACKAGE DECLARATION THAT HAS A PRIVATE PART IS WRITTEN AS FOLLOWS:

```
package Name_Of_Package is
     -- THE VISIBLE PART.
private
     -- THE PRIVATE PART.
end Name_Of_Package;
```

INSTRUCTOR NOTES

(PLUG IN THE APPROPRIATE DECLARATIONS AND COMMENTS AS GIVEN IN PREVIOUS SLIDES.)

POINT OUT THE TWO DECLARATIONS OF List_Type, IN THE VISIBLE PART AND IN THE PRIVATE PART.

INSTRUCTOR SHOULD MAKE THE FOLLOWING POINTS TO SET THE STAGE FOR THE NEXT SLIDE (EXERCISE).

OUTSIDE THE PACKAGE AND ALSO WITHIN THE PACKAGE DECLARATION PRIOR TO THE FULL DECLARATION OF List_Type, List_Type IS A PRIVATE TYPE. WITHIN THE PACKAGE AFTER THE FULL DECLARATION, List_Type DENOTES THE SAME TYPE, BUT IT IS NOW A RECORD TYPE AND HAS THE ADDITIONAL OPERATIONS APPROPRIATE FOR RECORD TYPES, SUCH AS SELECTION OF COMPONENTS. THESE ADDITIONAL OPERATIONS ARE NOT AVAILABLE OUTSIDE THE PACKAGE.

COMPLETE DECLARATION FOR PACKAGE List_Package

```
package List_Package is
   type Item_Type is ...;
   type Item_Array_Type is ...;
   type List_Type is private;
   Out_Of_Bounds : exception;
   function Length (...) return ...;
   function Array_To_List (...) return ...;
   function List_To_Array (...) return ...;
   function Element_Value (...) return ...;
   procedure Write_Element (...);
   procedure Insert (...);
   procedure Delete (...);
private
   Max_List_Length : constant := ...;
   subtype List_Storage_Space is Item_Array_Type (1 .. Max_List_Length);

   type List_Type is
      record
         Current_Length : Natural range 0 .. Max_List_Length := 0;
         Elements       : List_Storage_Space;
      end record;
end List_Package;
```

INSTRUCTOR NOTES

ANSWERS:

1. length	:=
Array_To_List	=
List_To_Array	/=
Element_Value	QUALIFIED EXPRESSIONS
Write_Element	
Insert	
Delete	

2. ALL OF THOSE IN #1 PLUS ADDITIONAL OPERATIONS APPROPRIATE FOR RECORD TYPES (EX. SELECTION OF COMPONENTS)

EXERCISE

1. WHAT OPERATIONS ARE AVAILABLE FOR List_Package FROM THE USER'S STANDPOINT?

2. WHAT OPERATIONS ARE AVAILABLE INSIDE THE PACKAGE, I.E., WHEN WRITING THE PACKAGE BODY?

INSTRUCTOR NOTES

NOTE THAT SINCE THESE FUNCTION BODIES WILL BE PLACED IN THE PACKAGE BODY FOR List_Type_Abstraction, TYPE List_Type IS A RECORD TYPE WITHIN THESE FUNCTIONS, I.E., THE HIDDEN REPRESENTATION OF ABSTRACT TYPE List_Type IS AVAILABLE WITHIN THESE FUNCTIONS.

ANSWER: RETURN List.Elements (1 .. List.Current_Length);

BODIES OF ARRAY-LIST CONVERSION FUNCTIONS

TO COMPLETE THE IMPLEMENTATION OF THE List_Type ABSTRACTION, WE MUST ALSO GIVE BODIES FOR THE FUNCTIONS Array_To_List AND List_To_Array.

```
function Array_To_List (Item_Array : Item_Array_Type) return List_Type is
                          -- YIELD A LIST OF THE ELEMENTS IN Item_Array
   List : List_Type;
begin
   if Item_Array'Length > Max_List_Length then
      raise Out_of_Bounds;
   end if;
   List.Elements (1 .. Item_Array'Length) := Item_Array;
   List.Current_Length := Item_Array'Length;
   return List;
end Array_To_List;
```

EXERCISE: COMPLETE THE IMPLEMENTATION:

```
function List_To_Array (List : List_Type) return Item_Array_Type is
                          -- YIELD AN ARRAY OF THE ELEMENTS IN List
begin -- List_To_Array



end List_To_Array;
```

INSTRUCTOR NOTES

EVEN THOUGH THE LANGUAGE RULES REQUIRE THE FULL TYPE DECLARATION TO BE A RECORD TYPE, OUTSIDE THE PACKAGE THAT DECLARES TYPE Varying_String_Type, THE TYPE IS STILL A PRIVATE TYPE, NOT A RECORD TYPE.

A COMPLETE ABSTRACTION AND IMPLEMENTATION FOR VARYING-LENGTH CHARACTER STRINGS (TYPE Varying_String_Type) IS GIVEN SUBSEQUENTLY IN SECTION 8.

PRIVATE TYPES CAN HAVE DISCRIMINANT PARTS

- IF A PRIVATE TYPE DECLARATION HAS A DISCRIMINANT PART, THE FULL TYPE DECLARATION MUST REPEAT THE DISCRIMINANT PART AND MUST DECLARE A RECORD TYPE.

- IF A PRIVATE TYPE DECLARATION HAS NO DISCRIMINANT PART, THE FULL TYPE MUST NOT BE AN UNCONSTRAINED TYPE WITH DISCRIMINANTS.

  EXAMPLE:

```
type Varying_String_Type (Max_Length : Natural) is private;
```

- PRIVATE TYPES THAT HAVE DISCRIMINANT PARTS MUST BE FULLY DECLARED AS RECORD TYPES.

  EXAMPLE:

```
type Varying_String_Type (Max_Length : Natural) is
   record
      Current_Length : Natural := 0;
      Content        : String (1 .. Max_Length);
   end record;
```

INSTRUCTOR NOTES

THE REASON FOR RULE 2 IS THAT DECLARING A VARIABLE OF THE PRIVATE TYPE WOULD NEED A CONSTRAINT (TO DETERMINE SIZE), BUT THE TYPE IS PRIVATE AND CANNOT BE GIVEN THE NEEDED INDEX CONSTRAINT.

IN RULE 3 THEY CAN BE USED IN TYPE OR SUBTYPE DECLARATIONS, SUBPROGRAM SPECIFICATIONS, DEFERRED CONSTANT DECLARATIONS, OR ENTRY DECLARATIONS. THE REASON FOR RULE 3 IS THAT ANY OTHER USE OF THE NAME REQUIRES KNOWING THE SIZE OR COMPOSITION OF THE PRIVATE TYPE (WHICH IS NOT YET FULLY DECLARED). IN PARTICULAR, RULE 3 PROHIBITS VARIABLES OF THE PRIVATE TYPE FROM BEING DECLARED IN THE VISIBLE PART OF THE PACKAGE DECLARATION.

SOME RESTRICTIONS ON THE DECLARATION AND USE OF PRIVATE TYPES:

1. A PRIVATE TYPE DECLARATION CAN ONLY BE GIVEN IN THE VISIBLE PART OF A PACKAGE DECLARATION. A FULL TYPE DECLARATION FOR THE PRIVATE TYPE MUST BE GIVEN IN THE CORRESPONDING PRIVATE PART OF THE PACKAGE DECLARATION.

2. THE FULL TYPE MUST NOT BE AN UNCONSTRAINED ARRAY TYPE.

3. WITHIN THE PACKAGE DECLARATION, BETWEEN THE PRIVATE AND FULL TYPE DECLARATIONS, THE NAMES OF THE TYPE, OR OF ITS SUBTYPES, OR OF ANY TYPES OR SUBTYPES THAT CONTAIN THE PRIVATE TYPE AS A SUBCOMPONENT CAN NOT BE USED WITHIN EXPRESSIONS.

INSTRUCTOR NOTES

NOTABLE POINTS

- USER ONLY INTERESTED IN "PUBLIC PART".

- REPRESENTATION MAY BE CHANGED WITHOUT INVALIDATING LOGIC OF PROGRAMS USING THE ABSTRACTION.

EXAMPLE:

  - CHANGE FROM LINEAR LIST TO LINKED LIST
  - OPERATIONS ARE LIST OPERATIONS (NOT ARRAY OR POINTER OPERATIONS)

INSTRUCTOR NOTES

THE REASON FOR THE LAST BULLET IS THAT ANY OTHER USE OF A DEFERRED CONSTANT NEEDS THE CONSTANT'S INITIAL VALUE (WHICH HASN'T BEEN GIVEN YET).

DEFERRED CONSTANTS

- CONSTANTS THAT ARE OF A PRIVATE TYPE ARE CALLED DEFERRED CONSTANTS.
- INITIAL VALUES CANNOT BE GIVEN UNTIL AFTER THE FULL TYPE DECLARATION FOR THE PRIVATE TYPE IS GIVEN IN THE PACKAGE'S PRIVATE PART.
- THE CORRESPONDING FULL CONSTANT DECLARATION (WITH AN EXPLICIT INITIALIZATION) MUST OCCUR AFTER THE FULL TYPE DECLARATION AND IN THE SAME PRIVATE PART.
- WITHIN THE PACKAGE DECLARATION, BETWEEN A DEFERRED CONSTANT DECLARATION AND ITS FULL CONSTANT DECLARATION, THE NAME OF THE DEFERRED CONSTANT CAN ONLY BE USED IN DEFAULT EXPRESSIONS FOR RECORD COMPONENTS OR FOR FORMAL PARAMETERS.

```
package List_Package is
   ...
   type List_Type is private;
   Null_List : constant List_Type;               -- A DEFERRED CONSTANT
   function Length (...) return ...;
   ...
private
   ...
   type List_Type is record ... end record;
   Null_List : constant List_Type := (0, (others =>  Junk_Item));
                                                 -- THE FULL DECLARATION
end List_Package;
```

INSTRUCTOR NOTES

SECTION 8

LIMITED PRIVATE TYPES

INSTRUCTOR NOTES

NOW WE ARE LOOKING AT THE PACKAGE FROM THE IMPLEMENTOR'S VIEW.

RECALL THAT A PRIVATE TYPE HAS PREDEFINED ASSIGNMENT AND EQUALITY OPERATIONS.

THE PREDEFINED := FOR A PRIVATE TYPE MERELY COPIES THE ENTIRE PRIVATE OBJECT, USING THE PREDEFINED := THAT APPLIES TO THE FULLY DECLARED PRIVATE TYPE.

REVIEW OF PREDEFINED OPERATIONS

AVAILABLE TO USERS OF PRIVATE TYPES

:=   =   /=

- DEPENDING ON THE IMPLEMENTATION OF THE TYPE IN THE PRIVATE PART, USE OF THE PREDEFINED OPERATIONS MAY LEAD TO UNEXPECTED RESULTS.

- PREDEFINED OPERATIONS ARE OFTEN NOT APPROPRIATE FOR THE DESIRED ABSTRACTIONS. IN PARTICULAR, THIS APPLIES TO ABSTRACTIONS FOR VARYING-LENGTH STRINGS, LISTS, STACKS, AND QUEUES.

- EXPLAIN THAT
  - <u>THE PREDEFINED :=</u> COPIES THE ENTIRE RECORD
    - INCLUDING THE ENTIRE List_Storage_Space ARRAY COMPONENT, I.E., ELEMENTS (1 .. Max_List_Length).
  - <u>THE PREDEFINED =</u> COMPARES THE ENTIRE RECORD
    - INCLUDING THE ENTIRE List_Storage_Space ARRAY COMPONENT, I.E., ELEMENTS (1 .. Max_List_Length).

- EXPAND ON THE EXPLANATION OF INCORRECTNESS ... THESE ELEMENTS COULD DIFFER IN THE TWO OBJECTS BEING COMPARED, THEREBY FORCING THE PREDEFINED = TO YIELD FALSE, EVEN WHEN ELEMENTS (1 .. Current_Length) ARE THE SAME IN BOTH OBJECTS AND WE WANT IT TO YIELD TRUE.

PREDEFINED OPERATIONS USING RECORD TYPE IMPLEMENTATION

CONTEXT:

```
package List_Package is
    .
    .
    .
private
   Max_List_Length : constant := ...;
   subtype List_Storage_Space is Item_Array_Type (1 .. Max_List_Length);
   type List_Type is
      record
         Current_Length : Natural range 0 .. Max_List_Length := 0;
         Elements       : List_Storage_Space;
      end record;
end List_Package;
```

THE PREDEFINED := COPIES THE ENTIRE RECORD

PROBLEM: INEFFICIENCY

- ONLY ELEMENTS (1 .. Current_Length) ARE MEANINGFUL AND NEED TO BE COPIED

THE PREDEFINED = COMPARES THE ENTIRE RECORD

PROBLEM: INCORRECTNESS

- ELEMENTS (Current_Length + 1 .. Max_List_Length) ARE JUNK AND MUST NOT BE COMPARED

INSTRUCTOR NOTES

PREDEFINED OPERATIONS USING LINKED LIST IMPLEMENTATION

CONTEXT:

```
type List_Cell;

type List_Type is access List_Cell;

type List_Cell is
  record
      Element   : Item_Type;
      Link_Part : List_Type;
  end record;
```

THE PREDEFINED := COPIES THE ACCESS VALUE

PROBLEM: INCORRECTNESS

- L1 := L2 CAUSES L1 TO SHARE L2'S LIST, RATHER THAN TO BE AN ELEMENT BY ELEMENT COPY OF L2

THE PREDEFINED = COMPARES ACCESS VALUES

PROBLEM: INCORRECTNESS

- YIELDS FALSE FOR TWO DIFFERENT ACCESS VALUES EVEN IF THEY POINT TO TWO LISTS THAT CONTAIN THE SAME CORRESPONDING INDIVIDUAL ELEMENT VALUES. (WE WANT TRUE.)

INSTRUCTOR NOTES

ASK CLASS TO GIVE EXAMPLES WHERE PRIVATE TYPES ARE APPROPRIATE (E.G., Counter, ...).

RECALL THAT A PRIVATE TYPE HAS PREDEFINED ASSIGNMENT AND EQUALITY OPERATIONS.

REVIEW PRIVATE TYPE OPERATIONS

- WHAT DO := AND = ACTUALLY DO FOR List_Type OBJECTS?

- IS IT WHAT WE REALLY WANT?

CONTEXT:

```
package List_Package is
     type Item_Type is ...;
     type Item_Array_Type is ...;
     type List_Type is private;
     Null_List : constant List_Type;
     function Length (...) return ...;
     ...
private
     Max_List_Length : constant := ...;
     subtype List_Storage_Space is Item_Array_Type (1 .. Max_List_Length);
     type List_Type is
       record
            Current_Length : Natural range 0 .. Max_List_Length := 0;
            Elements       : List_Storage_Space;
       end record;
     Null_List : constant List_Type := (0, (others => Junk_Item));
end List_Package;
```

INSTRUCTOR NOTES

- MEMBERSHIP (IN A SUBTYPE WITH A GIVEN DISCRIMINANT CONSTRAINT) AND TYPE CONVERSION ARE ALSO PREDEFINED FOR LIMITED PRIVATE TYPES.

- TYPE CONVERSION (ITEM NO. 2) REFERS TO DERIVING A TYPE FROM A LIMITED PRIVATE TYPE AND HAVING THE ABILITY TO CONVERT BETWEEN THE PARENT AND ITS DERIVED TYPE. DO NOT GO INTO DETAIL. SECTION 12 DEALS WITH DERIVED TYPES.

LIMITED PRIVATE TYPES

A LIMITED PRIVATE TYPE IS A PRIVATE TYPE THAT HAS NO PREDEFINED OPERATIONS (ASSIGNMENT (:=), EQUALITY (=), OR INEQUALITY (/=)).

```
package List_Package is
     ...
     type List_Type is limited private;

     function Length (...) return ...;
     ...
private
     ...
     type List_Type is ...;   -- FULL DECLARATION FOR List_Type.
     ...
end List_Package;
```

OPERATIONS AVAILABLE TO A USER OF List_Type:

1. THOSE THAT ARE EXPLICITLY DECLARED AS SUBPROGRAMS IN THE VISIBLE PART OF THE PACKAGE SPECIFICATION, E.G., Length.

2. THOSE THAT ARE PREDEFINED FOR LIMITED PRIVATE TYPES, SUCH AS QUALIFIED EXPRESSIONS, SUBTYPE MEMBERSHIP, AND TYPE CONVERSION.

INSTRUCTOR NOTES

SUBPROGRAM BODIES FOR Copy AND Equal ARE SHOWN ON THE NEXT TWO SLIDES.

ASSIGNMENT AND EQUALITY

ADDITIONAL SUBPROGRAMS MUST BE SPECIFIED IN THE VISIBLE PART OF PACKAGE List_Type ABSTRACTION TO PROVIDE APPROPRIATE ASSIGNMENT AND EQUALITY OPERATIONS THAT SATISFY OUR DESIRED ABSTRACTION.

```
procedure Copy (Source : in List_Type; Target : out List_Type);
               -- ASSIGN SOURCE LIST Source TO TARGET LIST Target.
               -- (INDIVIDUAL ELEMENTS ARE COPIED.)

function Equal (Left, Right : List_Type) return Boolean;
               -- COMPARE LISTS Left AND Right FOR EQUALITY.
               -- (INDIVIDUAL ELEMENTS ARE COMPARED.)
```

THESE OPERATIONS MAY THEN BE USED AS FOLLOWS:

```
L1 : List_Type;
L2 : List_Type;
...
if not Equal (L1, L2) then
    Copy (L1, L2);
end if;
```

INSTRUCTOR NOTES

CHECK THAT THE BODY SATISFIES ITS COMMENTED SPECIFICATION.

PROCEDURE COPY

THE APPROPRIATE SUBPROGRAM BODIES FOR Copy AND Equal MUST ALSO BE ADDED TO THE BODY OF List_Package.

```
procedure Copy (Source : in List_Type; Target : out List_Type) is
            -- ASSIGN SOURCE LIST Source TO TARGET LIST Target.
            -- (INDIVIDUAL ELEMENTS ARE COPIED.)
begin -- Copy
    Target.Elements (1 .. Source.Current_Length) :=
         Source.Elements (1 .. Source.Current_Length);
    Target.Current_Length := Source.Current_Length;
end Copy;
```

NOTE THAT WITHIN THE BODY OF Copy, WE COULD ALSO MERELY ASSIGN THE ENTIRE RECORD OBJECT, I.E.,

```
Target := Source;
```

SINCE WE ARE WITHIN THE PACKAGE BODY OF List_Package AND IN HERE TYPE List_Type IS A RECORD TYPE, NOT A LIMITED PRIVATE TYPE, AND HAS A PREDEFINED := OPERATION. BUT THIS WILL COPY THE JUNK ELEMENTS AND THUS IS LESS EFFICIENT.

INSTRUCTOR NOTES

CHECK THAT THE BODY SATISFIES ITS COMMENTED SPECIFICATION.

ANSWER:

```
return Left.Elements (1 .. Left.Current_Length) =
       Right.Elements (1 .. Right.Current_Length);
```

EXERCISE: FUNCTION EQUAL

COMPLETE THE FOLLOWING:

```
function Equal (Left, Right : List_Type) return Boolean is
            -- COMPARE LISTS Left AND Right FOR EQUALITY.
            -- (INDIVIDUAL ELEMENTS ARE COMPARED.)
begin -- Equal



end Equal;
```

REMEMBER THAT = FOR SLICES YIELDS False IF THE LENGTHS ARE DIFFERENT.

NOTE THAT EVEN THOUGH THE = OPERATION FOR RECORD TYPE List_Type IS DEFINED WITHIN THE BODY OF Equal, WE CANNOT MAKE USE OF IT BECAUSE IT ALSO COMPARES THE JUNK ELEMENTS AS EXPLAINED PREVIOUSLY.

INSTRUCTOR NOTES

TASK TYPES AND TYPES DERIVED FROM LIMITED TYPES ARE ALSO LIMITED.

ANSWERS:
```
-- ILLEGAL; LIMITED TYPE
-- LEGAL
-- ILLEGAL; LIMITED TYPE
-- LEGAL
-- ILLEGAL; LIMITED TYPE
-- LEGAL
-- ILLEGAL; LIMITED TYPE
-- LEGAL
```

OF COURSE, ONE CAN ALWAYS DECLARE SUBPROGRAMS SUCH AS Copy AND Equal TO PERFORM THESE OPERATIONS FOR LIMITED COMPOSITE TYPES. FOR EXAMPLE:

```
type Personal_Data is
   record
        Age : Natural;
        Jobs_Held : List_Type; -- Component type is limited.
   end record;

procedure Copy_Personal_Data (Source : in Personal_Data;
                              Target : in out Personal_Data) is
begin -- Copy_Personal_Data
   Target.Age := Source.Age;
   Copy (Source.Jobs_Held, Target.Jobs_Held);
end Copy_Personal_Data;

function Equal_Personal_Data (Left, Right : Personal_Data) return Boolean is
begin -- Equal_Personal_Data
   return Left.Age = Right.Age and Equal (Left.Jobs_Held, Right.Jobs_Held);
end Equal_Personal_Data;
```

LIMITED COMPOSITE TYPES

- A COMPOSITE TYPE IS LIMITED IF THE TYPE OF ANY OF ITS SUBCOMPONENTS IS LIMITED.

- FOR A LIMITED COMPOSITE TYPE, THERE ARE :=, =, AND /= OPERATIONS FOR THOSE INDIVIDUAL COMPONENTS WHOSE TYPES ARE NOT LIMITED.

EXAMPLE:

```
    type Personal_Data is
           record
                Age : Natural;
                Jobs_Held: List_Type;                        -- COMPONENT TYPE IS
           end record;                                       -- LIMITED, HENCE THE
                                                             -- ENTIRE RECORD
                                                             -- TYPE IS LIMITED

    Person_1, Person_2 : Personal_Data;
    Same_Person : Boolean;
```

COMMENT WHETHER THE FOLLOWING ARE LEGAL OR ILLEGAL:

```
Person_1 := Person_2;                                              --
Person_1.Age := Person_2.Age;                                      --
Person_1.Jobs_Held := Person_2.Jobs_Held;                          --
Copy (Person_2.Jobs_Held, Person_1.Jobs_Held);                     --

Same_Person := Person_1 = Person_2;                                --
Same_Person := Person_1.Age = Person_2.Age;                        --
Same_Person := Person_1.Jobs_Held = Person_2.Jobs_Held;            --
Same_Person := Equal (Person_1.Jobs_Held, Person_2.Jobs_Held);     --
```

INSTRUCTOR NOTES

THE DEFERRED CONSTANT DECLARATION IS PERMITTED BECAUSE := EXISTS FOR THE FULL TYPE (THE RECORD TYPE).

THE FORMAL PARAMETER DEFAULT EXPRESSION IS PERMITTED BECAUSE NO ASSIGNMENT IS INVOLVED IN A SUBPROGRAM CALL; THE DEFAULT VALUE IS MERELY USED AS THE ACTUAL PARAMETER WHEN NO EXPLICIT ACTUAL PARAMETER IS GIVEN IN THE CALL.

THE FOLLOWING USES OF LIMITED TYPES ARE PERMITTED:

1. A DEFERRED CONSTANT MAY BE OF A LIMITED PRIVATE TYPE IF THE FULL TYPE IS NOT LIMITED.

```
package List_Package is
     ...
     type List_Type is limited private;
     Null_List : constant List_Type;
     ...
private
     ...
     type List_Type is record ... end record;
     Null_List : constant List_Type := ...;
end List_Package;
```

2. A FORMAL PARAMETER OF MODE <u>in</u> AND OF A LIMITED TYPE MAY HAVE A DEFAULT EXPRESSION.

```
procedure P (List : List_Type := Null_List);
```

INSTRUCTOR NOTES

THE REASON FOR THESE RESTRICTIONS IS THAT A LIMITED TYPE HAS NO PREDEFINED := OPERATION.

THE PROHIBITION OF := IS REALLY A PROHIBITION OF SEVERAL OPERATIONS THAT REQUIRE COPYING.

THE INSTRUCTOR SHOULD COME UP WITH EXAMPLES TO DEMONSTRATE THESE POINTS.

SPECIAL CONSIDERATIONS WHEN USING Limited Private TYPES

THE EFFECT OF THE DESIGNER HAVING COMPLETE CONTROL OVER WHICH OPERATIONS ARE AVAILABLE TO THE USER:

- UNAVAILABILITY OF ASSIGNMENT OPERATOR (:=)
  - THE DECLARATION OF AN OBJECT CANNOT INCLUDE AN INITIAL VALUE
  - A CONSTANT CANNOT BE DECLARED OUTSIDE THE DEFINING PACKAGE
  - A RECORD COMPONENT OF A LIMITED TYPE CANNOT HAVE A DEFAULT INITIAL EXPRESSION

- A FORMAL PARAMETER WHOSE TYPE IS LIMITED CANNOT BE OF MODE out UNLESS THE TYPE IS LIMITED PRIVATE AND THE SUBPROGRAM IS DECLARED IN THE VISIBLE PART OF THE PACKAGE THAT DECLARES THE TYPE. IF MODE out IS USED FOR SUCH A FORMAL PARAMETER, THE CORRESPONDING FULL TYPE MUST NOT BE LIMITED.

INSTRUCTOR NOTES

REVIEW OF ABSTRACTION

- ABSTRACTION RELIES HEAVILY ON THE USE OF Private AND Limited Private TYPES IN ORDER TO:
  - GIVE THE IMPLEMENTOR MAXIMUM FREEDOM
  - MAINTAIN THE INTEGRITY AND CONSISTENCY OF THE ABSTRACTION

- PRIVATE TYPES
  - PRIMARY MECHANISM FOR CREATING ABSTRACT DATA TYPES
  - DIRECTLY SUPPORT THE PRINCIPLES OF INFORMATION HIDING (IN WHICH THE DETAILS OF AN IMPLEMENTATION ARE SUPPRESSED IN ORDER TO FOCUS ON THE ABSTRACTION)
  - VALUES AND PREDEFINED OPERATIONS ARE HIDDEN FROM THE USER; ONLY EXPLICITLY NAMED OPERATIONS ARE VISIBLE
  - BY RESTRICTING ACCESSIBILITY TO DATA, A PACKAGE USING A Private or Limited Private TYPE IS SAID TO ENCAPSULATE THE DATA TYPE

INSTRUCTOR NOTES

Ada'S PREDEFINED STRING TYPE IS FOR FIXED LENGTH STRINGS. A VARYING-LENGTH STRING TYPE IS USEFUL IN MANY APPLICATIONS.

THIS Varying_String_Type IS SIMILAR TO PL/I'S CHARACTER (N) VARYING TYPE.

THE POINT BEING MADE HERE IS THAT EVEN VERY SIMPLE TYPES OR ENTITIES SHOULD SOMETIMES BE TREATED AS AN ABSTRACTION. VARYING LENGTH STRINGS ARE NOT ONLY REUSABLE ACROSS MANY APPLICATIONS BUT ALSO ERROR PRONE IF NOT MANIPULATED CORRECTLY. THEY ARE AN EXCELLENT CANDIDATE FOR AN ABSTRACTION.

AN ABSTRACTION IS OFTEN COMMON TO A NUMBER OF APPLICATIONS

EXAMPLE: Varying Strings

APPLICATIONS: ELECTRONIC MAIL SYSTEM

MESSAGE SWITCHING

DBMS QUERY SYSTEM

PERSONNEL RECORD

TEXT PROCESSING

AN ABSTRACTION MAY BE AN UNDERLYING "UNIT OF WORK" IN AN APPLICATION. THE PROGRAM SHOULD TREAT EACH INSTANCE UNIFORMLY, CONSISTENTLY, AND CORRECTLY.

INSTRUCTOR NOTES

HERE WE ARE TRYING TO GET THE <u>BARE BONES</u> ABSTRACTION.

DISCUSS THE VALUES, OPERATIONS, AND RELATIONSHIPS OF THE ABSTRACTION.

AT THE END OF THE SECTION WE WILL DISCUSS ENHANCEMENTS TO THIS PACKAGE SPECIFICATION.

INTRODUCE SUBPROGRAM NAME AS YOU DISCUSS THE DESIRED OPERATIONS.

LATER WE WILL SEE THAT ASSIGNMENT BECOMES Procedure Copy_String.

VARYING STRING ABSTRACTION

- VALUES
  - A SINGLE VARIABLE CAN HOLD A STRING OF ARBITRARY LENGTH
  - CONSIDER AN UPPER BOUND

- OPERATIONS
  - CREATION (Varying_String)
  - TERMINATION (Dispose_String)
  - CONVERSION (Varying_String, String_Content)
  - STATE INQUIRY (Substring, Length)
  - Input/Output REPRESENTATION (Text_IO WITH Varying_String CONVERSION SUBPROGRAMS)
  - STATE CHANGE (Catenate AND ALSO ANY ASSIGNMENTS)

- RELATIONSHIPS
  - EQUALITY

INSTRUCTOR NOTES

ANSWERS:

```
type Varying_String_Type (Capacity : Natural) is limited private;
  ...
  type Varying_String_Type (Capacity : Natural) is
  record
     Current_Length : Natural := 0;
     Content        : String (1 .. Capacity);
  end record;
end Varying_String_Package;
```

THE FOLLOWING SLIDES WILL HAVE STUDENTS WORK ON ALTERNATE IMPLEMENTATIONS OF THE PRIVATE PART.

THE EXCEPTION Too_Long WILL BE RAISED IF THE Capacity DISCRIMINANT COMPONENT OF THE TARGET Varying_String_Type OBJECT IS NOT LARGE ENOUGH FOR THE OBJECT TO HOLD THE DESIRED ABSTRACT STRING VALUE. THE EXCEPTION Invalid_Position WILL BE RAISED IF, FOR EXAMPLE, THE POSITION PARAMETER From DOES NOT DESCRIBE A VALID POSITION WITHIN THE Varying_String_Type OBJECT. IF AN ACTUAL PARAMETER OF VALUE < 1 WERE PASSED AS A POSITION, Ada WOULD REQUIRE THAT THE PREDEFINED EXCEPTION Constraint_Error BE RAISED. IN ORDER TO ISOLATE ERRORS CAUSED BY POSITION PARAMETERS NOT IN THE PROPER RANGE, IT IS BETTER TO RAISE THE USER-DEFINED EXCEPTION Invalid_Position WHEN A POSITION IS IN THE NATURAL RANGE BUT IS GREATER THAN THE CURRENT LENGTH OF THE Varying_String_Type OBJECT + 1, I.E., IS PAST THE END OF THE Varying_String_Type OBJECT'S ABSTRACT STRING VALUE.
THE TWO IMPORTANT DECLARATIONS ARE: TYPE Varying_String_Type AND THE CONSTANT Empty. NOTE THAT THERE IS NO DEFAULT VALUE FOR THE Capacity DISCRIMINANT COMPONENT OF TYPE Varying_String_Type. THIS IS BECAUSE: 1) THERE IS NO ONE DEFAULT VALUE THAT IS MEANINGFUL FOR MOST APPLICATIONS; and 2) USE OF THE DEFAULT WOULD CAUSE THE MAXIMUM SPACE (FOR Max_Capacity CHARACTERS) TO BE ALLOCATED, AND WOULD PROBABLY RAISE Storage_Error.
Ada'S LINEAR ELABORATION RULES REQUIRE THAT THE DECLARATIONS BE IN THE ORDER SHOWN IN THE SLIDE.
A PERFECTLY REASONABLE ALTERNATIVE IS A LINKED LIST OF CHARACTERS. WE WILL CONTINUE WITH THE RECORD-WITH-DISCRIMINANTS IMPLEMENTATION.

EXERCISE

Varying_String_Package SPECIFICATION

FILL IN THE TYPE DECLARATION AND COMPLETE THE PRIVATE SECTION AS NECESSARY:

```
package Varying_String_Package is

   type            (       :       ) is             ;
   procedure Copy_String(From: in Varying_String_Type;
                         To  : out Varying_String_Type);
   procedure Dispose_String (Varying_String : Varying_String_Type);
   function Catenate (Left, Right : Varying_String_Type) return Varying_String_Type;
   function Substring (Varying_String : Varying_String_Type; From, To : Natural)
                              return Varying_String_Type;
   function Same_String (Left, Right : Varying_String_Type) return Boolean;
   function Varying_String (S : String) return Varying_String_Type;
   function String_Content (Varying_String : Varying_String_Type) return String;
   function Length (Varying_String : Varying_String_Type) return Natural;
   Too_Long : exception;
   Invalid_Position : exception;

private



end Varying_String_Package;
```

INSTRUCTOR NOTES

ALLOW THE STUDENTS SEVERAL MINUTES TO THINK ABOUT AN ALTERNATE IMPLEMENTATION AND THEN PROVIDE IT FOR THEM.

IN THIS VERSION OF TYPE Varying_String_Type, THE TYPE HAS NO DISCRIMINANT. THE SAME MAXIMUM LENGTH, NAMELY, Capacity, APPLIES TO ALL Varying_String_Type OBJECTS.

Varying_String_Type OBJECTS ARE DYNAMICALLY ALLOCATED. IF THE CURRENT SPACE ALLOCATED FOR A Varying_String_Type OBJECT IS NOT LARGE ENOUGH TO HOLD ITS NEW VALUE, THEN SUFFICIENT NEW SPACE IS ALLOCATED FOR THE NEW VALUE, AND THE OLD SPACE IS FREED.

THIS VERSION OF TYPE Varying_String_Type IS USED IN THE Ada COMPILER VALIDATION CAPABILITY TOOLS.

```
package Varying_String_Package is
     type Varying_String_Type is limited private;
     ...
private
     type Text_Type is access String;
                         -- Varying_String_Type CONTENT IS DYNAMICALLY ALLOCATED.
     type Varying_String_Type is
          record
             Current_Length : Varying_String_Length_Type := 0;       -- CURRENT LENGTH.
             Content : Text_Type := new String (0 .. 0);             -- CURRENT CONTENT.
          end record;
end Varying_String_Package;
```

EXERCISE

PROVIDE AN ALTERNATIVE Varying_String_Type ABSTRACTION

INSTRUCTOR NOTES

AN OVERVIEW OF THE PACKAGE BODY IS SHOWN. CODED SUBPROGRAMS ARE ON THE FOLLOWING SLIDES.

Varying_String_Package BODY

```
package body Varying_String_Package is

   procedure Copy_String (...) is ... begin ... end Copy_String;
   procedure Dispose_String (...) is ... begin ... end Dispose_String;
   function Catenate (...) return ... is ... begin ... end Catenate;
   function Substring (...) return ... is ... begin ... end Substring;
   function Same_String (...) return ... is ... begin ... end Same_String;
   function Varying_String (...) return ... is ... begin ... end Varying_String;
   function String_Content (...) return ... is ... begin ... end String_Content;
   function Length (...) return ... is ... begin ... end Length;
end Varying_String_Package;
```

INSTRUCTOR NOTES

NOTE THAT BECAUSE Varying_String_Type IS A LIMITED TYPE, IT HAS NO PREDEFINED := OPERATION, SO THESE PROCEDURES ARE NEEDED.

THE EFFECT COMMENT DEFINES THE EFFECT OF INVOKING THE PROCEDURE BY DEFINING THE NEW VALUES OF THE AFFECTED STATE FUNCTIONS (NAMELY, Content) IN TERMS OF THE OLD VALUES OF THE STATE FUNCTIONS AND FORMAL PARAMETERS.

ALTHOUGH THE EXCEPTION CONDITIONS ARE SPECIFIED LAST, IT IS TO BE UNDERSTOOD THAT THEY ARE CHECKED BEFORE ANY Varying_String_Type OBJECTS ARE ALTERED, JUST AS Ada'S PREDEFINED STRING := CHECKS LENGTHS BEFORE ALTERING THE TARGET STRING OBJECT.

THE COMMENT THAT Target MAY OVERLAP Source IN THE DECLARATION OF Copy_String MEANS THAT IT IS PERMISSIBLE FOR FORMAL PARAMETERS Source AND Target TO DENOTE THE SAME ACTUAL Varying_String_Type (VARIABLE) OBJECT. IN OTHER WORDS, THE IMPLEMENTATION OF Copy_String MUST BE CAREFUL TO AVOID THE ERRONEOUS SITUATIONS THAT CAN OCCUR WHEN ALIASING OF PRIVATE OR AGGREGATE PARAMETERS IS INVOLVED. SINCE Ada ALLOWS OVERLAP FOR THE PREDEFINED STRING :=, IT IS DESIRABLE TO ALSO ALLOW IT FOR Varying_String_Type Assign.

POINT OUT Dispose_String IS PROVIDED SO THAT STORAGE CAN BE RECLAIMED.

Varying_String_Package BODY

```
procedure Copy_String (From : in Varying_String_Type;
                       To   : in out Varying_String_Type) is
          -- ASSIGNS Varying String (From) TO Varying String (To)

begin
   if From.Current_Length > To.Capacity then
      raise Too_Long;
   end if;
   To.Content(1 .. From.Current_Length) := From.Content (1 .. From.Current_Length);
   To.Current_Length := From.Current_Length;
end Copy_String;

procedure Dispose_String (Varying_String : in Varying_String_Type) is
          -- TERMINATE A Varying String

begin
   Varying String.Current_Length := 0;
end Dispose_String;
```

INSTRUCTOR NOTES

REMIND STUDENTS THE VARYING STRING TYPE WITHIN THE PACKAGE BODY IS VIEWED AS A RECORD WITH A DISCRIMINANT COMPONENT WHICH HAS A DEFAULT VALUE THEREFORE:

- WHEN AN OBJECT OF Varying_String_Type IS DECLARED IN THE PACKAGE BODY, A DISCRIMINANT IS OPTIONAL
- PROVIDED THAT A DISCRIMINANT CONSTRAINT IS NOT USED WHEN DECLARING THE OBJECT, THEN THE VALUE OF THE DISCRIMINANT MAY BE CHANGED THROUGH WHOLE RECORD ASSIGNMENT

THIS IS A FREQUENTLY NEEDED OPERATION FOR VARYING-LENGTH STRINGS.

IN THE Catenate FUNCTION, THE LENGTH CHECK IS PERFORMED THIS WAY TO AVOID RAISING Constraint_Error OR Numeric_Error IF THE SUM OF THE LENGTHS EXCEEDED Natural'Last.

Varying_String_Package BODY

```
function Catenate (Left, Right : Varying_String_Type)
                  return Varying_String_Type is
         -- CATENATES LEFT WITH RIGHT AND RETURNS THE NEW VARYING LENGTH STRING

   Target : Varying_String_Type;

begin
   if Left.Current_Length > Natural'Last - Right.Current_Length then
      raise Too_Long;
   end if;

   Target := (Capacity | Current_Length => Left.Current_Length + Right.Current_Length;
                         Content => Left.Content & Right.Content);
   return Target;
end Catenate;
```

INSTRUCTOR NOTES

IN THE Substring FUNCTION, CAPACITY IS NOT ASSIGNED THE ACTUAL LENGTH OF THE SUBSTRING IN CASE To-From WERE NEGATIVE.

Varying_String_Package BODY

```
function Substring (Varying_String : Varying_String_Type; From, To : Natural);
                   return Varying_String_Type;
         -- EXTRACT A SUBSTRING FROM A Varying_String_Type
         -- BY SPECIFYING From, To AND RETURNING IT AS A
         -- Varying_String_Type
   Target : Varying_String_Type;

begin
   if From > Varying_String.Current_Length OR To > Varying_String.Current_Length then
      raise Invalid_Position;
   end if;

   Target := (Capacity => Varying_String.Capacity, Current_Length => To-From +1,
              Content => Varying_String.Content (From .. To));
   return Target;

end Substring;
```

INSTRUCTOR NOTES

POINT OUT THAT NO LENGTH CHECK IS MADE HERE BECAUSE THE DISCRIMINANT SUBTYPE IS Natural, WHOSE LARGEST VALUE, Integer'Last, IS THE SAME AS THAT OF Positive, THE INDEX SUBTYPE OF String. IF THE DISCRIMINANT SUBTYPE'S LARGEST VALUE WERE LESS THAN Integer'Last, THE FOLLOWING CHECK WOULD BE REQUIRED FOR DEFENSIVE PROGRAMMING:

```
if S'Length > Discriminant_Subtype'Last then
   raise Too_Long;
end if;
```

Varying_String_Package BODY

```
function Varying_String (S : String) Return Varying_String_Type is
         -- ACCEPTS A STRING AND RETURNS A Varying String
                                          --
   Target : Varying_String_Type;

begin
   Target := (Capacity | Current_Length => S'Length,
              Content => S);
   return Target;

end Varying_String;


function String_Content (Varying String : Varying_String_Type) return String is
         -- ACCEPTS A Varying String AND RETURNS A String

begin
   return Varying_String.Content;
end String_Content;


function Length (Varying_String : Varying_String_Type) return Natural is
         -- RETURNS THE LENGTH OF A Varying String TYPE

begin
   return Varying_String.Current_Length;
end Length;
```

INSTRUCTOR NOTES

THE FOLLOWING SLIDES PROVIDE AN ENHANCED Varying_String_Package WHICH PROVIDES SOME OF THE FOLLOWING:

ANSWERS:

VALUES:

Empty_String

OPERATIONS:

I/O Representation (Get, Put)

Change - GIVEN A STRING OF AN ARBITRARY LENGTH. THIS OPERATION TAKES A SUBSTRING OF THAT STRING AND REPLACES IT WITH ANOTHER STRING OF ARBITRARY LENGTH

Next

RELATIONSHIPS:

<, >, <=, >=

EXERCISE

WHAT OTHER VALUES, OPERATIONS, OR RELATIONSHIPS MIGHT BE USEFUL TO ADD TO A Varying_String ABSTRACTION?

VALUES

OPERATIONS

RELATIONSHIPS

INSTRUCTOR NOTES

THIS IS A GENERAL CHANGE OPERATION FOR Varying_String_Type OBJECTS THAT REPLACES A SUBSTRING OF A TARGET OBJECT Target, THE SUBSTRING BEGINNING AT POSITION From AND BEING OF LENGTH Count, WITH A NEW STRING Source OF A POSSIBLY DIFFERENT LENGTH.

```
Length,
Next : state inquiry
Less_Than,
Greater_Than,
Less_Than_Or_Equal,
Greater_Than_Or_Equal : relationships
```

THIS IMPLEMENTATION OF next USES THE SIMPLE METHOD OF MATCHING ALL OF Content (Pattern) AGAINST SUCCESSIVE SUBSTRINGS OF Source OF LENGTH Length (Pattern). IN PRACTICE THIS IS USUALLY A LINEAR TIME ALGORITHM, ALTHOUGH THE WORST CASE TIME, WHICH IS O (Length (Pattern) * Length (Source)), CAN OCCUR FOR REPETITIVE PATTERNS SUCH AS Pattern = "AAAAB" IN REPETITIVE SOURCE STRINGS SUCH AS Source = "AAAAAAAAAAAAAAB". THERE ARE FASTER ALGORITHMS WITH LINEAR WORST CASE TIMES.

POINT OUT Text_IO WOULD BE PROVIDED FOR THE BODY OF PACKAGE Varying_String_Package TO SUPPORT I/O PROCEDURES Get_Line AND Put

ENHANCED Varying_String_Package

```
package Varying_String_Package is
   type Varying_String_Type is limited private;
   Empty : constant Varying_String_Type;

   procedure Copy_String         (From : in Varying_String_Type; To : out Varying_String_Type);
   procedure Dispose_String      (Varying_String : In Varying_String_Type);
   procedure Get_Line            (Varying_string : out Varying_String_Type);
   procedure Put                 (Varying_String : in Varying_String_Type);
   procedure Change              (Target : in out Varying_String_Type;
                                  Source : in Varying_String_Type; From, Count : in Natural);
   function Catenate             (Left, Right : Varying_String_Type) return Varying_String_Type;
   function Substring            (Varying_String : Varying_String_Type; From, To : Natural)
                                  return Varying_String_Type;
   function Same_String          (Left, Right : Varying_String_Type) return Boolean;
   function Varying_String       (S : String) return Varying_String_Type;
   function String_Content       (Varying_String : Varying_String_Type) return String;
   function Length               (Varying_String : Varying_String_Type) return Natural;
   function Next                 (Pattern, Source : Varying_String_Type; From : Natural)
                                  return Natural;
   function Less_Than            (Left, Right : Varying_String_Type) return Boolean;
   function Greater_Than         (Left, Right : Varying_String_Type) return Boolean;
   function Less_Than_Or_Equal   (Left, Right : Varying_String_Type) return Boolean;
   function Greater_Than_Or_Equal (Left, Right : Varying_String_Type) return Boolean;
   ...
private
   type Text_Type is access String;      -- Varying_String_Type CONTENT IS DYNAMICALLY ALLOCATED
   Empty_Text : constant Text_Type := new String (1 ..0);
                                         -- THE ONLY VALUE SHARED BY Varying_String_Type OBJECTS

      type Varying_String_Type is
             record
                Current_Length : Varying_String_Length_Type := 0;     -- CURRENT LENGTH
                Content : Text_Type := Empty_text;                    -- CURRENT CONTENT
             end record;
      Empty : constant Varying_String_Type := (0, Empty_Text);

end Varying_String_Package;
```

INSTRUCTOR NOTES

MESSAGE PROCESSING APPLICATION

- PROBLEM STATEMENT:

  - THE MESSAGE PROCESSOR ACCEPTS A MESSAGE AND SCANS IT FOR A PRIORITY KEY AND A DESTINATION KEY.

  - IF THE MESSAGE IS TOP PRIORITY, CATENATE A STANDARD ACKNOWLEDGEMENT TO THE END OF THE MESSAGE. IF NOT, CATENATE A GENERAL CLEARANCE MESSAGE.

  - REPLACE THE DESTINATION KEY WITH THE COMPLETE ADDRESS FOUND IN THE ADDRESS DATABASE.

INSTRUCTOR NOTES

THIS SLIDE IS SETTING UP THE CONTEXT FOR THE NEXT SLIDE.

BRIEFLY GO OVER AND POINT OUT THAT SOME SORT OF DATABASE NEEDS TO BE PROVIDED AS WELL AS CAPABILITIES TO READ ADDRESSES FROM IT, WRITE ADDRESSES, AND INSERT ADDITIONAL ADDRESSES. THIS IS QUITE SIMPLIFIED.

POINT OUT THAT THE STRUCTURE OF THE DATABASE ITSELF IS HIDDEN INSIDE THE PACKAGE BODY.

POINT OUT THE USE OF THE SUBTYPE DECLARATION TO RENAME Varying_String_Type SO THE EXPANDED (QUALIFIED) NAME DOES NOT NEED TO BE USED.

MESSAGE PROCESSING APPLICATION

CONTEXT:

```
with Varying_String_Package;
with Text_IO; use Text_IO;
package Address_Database_Package is
   subtype Varying_String_Type is Varying_String_Package.Varying_String_Type;
   function Read_Address (Address : in Varying_String_Type) return
                                             Varying_String_Type;
   procedure Write_Address (Address : in Varying_String_Type);
   procedure Append_Address (Address : in Varying_String_Type);
end Address_Database_Package;
```

INSTRUCTOR NOTES

BE SURE TO POINT OUT THE USE OF Copy_String (:= is illegal). LIKEWISE THE USE OF Substring ("message (1 .. 9)" is illegal), ETC...

WALK THROUGH CODE CAREFULLY.

MESSAGE PROCESSING APPLICATION

```
with Address_Database_Package;
with Varying_String_Package; use Varying_String_Package;
procedure Message_Processing (Message : in out Varying_String_Type) is
   function Get_Address (Address : Varying_String_Type) return Varying_String_Type
                    renames Address_Database_Package.Read_Address;
   Top_Secret, Clearance, Address : Varying_String_Type;
begin
   if String_Content (Substring (Message, 1, 1)) = "1"
      then Copy_String (From => Catenate (Message, Varying_String ("top secret")),
                        To   => Message);
   end if;

   Copy_String (Get_address (Substring (Message, 10, 12 )), Address);
   Copy_String (Catenate (Catenate (Substring (Message, 1, 9), Address),
                Substring (Message, 13, Length (Message))), Message);
end Message_Processing;
```

INSTRUCTOR NOTES

SECTION 9

USE OF EXCEPTIONS

INSTRUCTOR NOTES

ANSWER TO THE QUESTION IN THE TITLE: Pop RAISES Constraint_Error.

SINCE Off_Of.Top_Part IS ZERO, THE INDEX VALUE IN Onto.Contents (Onto.Top_Part) IS OUT OF BOUNDS. THEREFORE, THE FIRST ASSIGNMENT STATEMENT RAISES Constraint_Error. SINCE Pop HAS NO HANDLER FOR Constraint_Error, THE EXCEPTION IS PROPAGATED TO THE CALL ON Pop.

SIMILARLY, IF Push HAD BEEN CALLED WITH A FULL STACK, SO THAT Onto.Top_Part = 100, THE FIRST ASSIGNMENT STATEMENT IN THE Push PROCEDURE BODY WOULD VIOLATE THE RANGE CONSTRAINT IN THE Top_Part RECORD COMPONENT DECLARATION. THIS TOO WOULD RAISE Constraint_Error.

WHAT HAPPENS WHEN Pop IS CALLED WITH AN EMPTY STACK?

```
package Stack_Package is

   type Stack_Type is limited private;

   procedure Push (Item : in Integer; Onto : in out Stack_Type);
   procedure Pop  (Item : out Integer; Off_Of : in out Stack_Type);

private

   type Integer_List_Type is array (1 .. 100) of Integer;

   type Stack_Type is
      record
         Top_Part      : Integer range 0 .. 100 := 0;  -- Initially empty
         Contents_Part : Integer_List_Type;
      end record;

end Stack_Package;

package body Stack_Package is

   procedure Push (Item : in Integer; Onto : in out Stack_Type) is
   begin
      Onto.Top_Part                     := Onto.Top_Part + 1;
      Onto.Contents_Part (Onto.Top_Part) := Item;
   end Push;

   procedure Pop (Item : out Integer; Off_Of : in out Stack_Type) is
   begin
      Item            := Off_Of.Contents_Part (Off_Of.Top_Part);
      Off_Of.Top_Part := Off_Of.Top_Part - 1;
   end Pop;

end Stack_Package;
```

INSTRUCTOR NOTES

THE APPARENTLY INFINITE LOOP IS MEANT TO BE EXITED WHEN Constraint_Error IS RAISED BY THE CALL ON Pop. THEN CONTROL IS AUTOMATICALLY PASSED TO THE HANDLER.

AS THE NEXT FEW SLIDES WILL EXPLAIN, THIS IS NOT GOOD PROGRAMMING STYLE. WE ARE PRESENTING IT AS A COUNTEREXAMPLE.

USING THIS VERSION OF Stack_Package

A LOOP TO POP ALL REMAINING VALUES OFF THE STACK AND PRINT THE AVERAGE OF THE VALUES

POPPED:

```
with Stack_Package, Text_IO;

procedure Pop_And_Average (Stack : in out Stack_Package.Stack_Type) is
    package Type_Float_IO is new Text_IO.Float_IO (Float);
    use Type_Float_IO;

    Number_Popped : Integer := 0;
    Sum           : Integer range 0 .. 10_000 := 0;
    Item          : Integer;

begin

    loop
        Stack_Package.Pop (Item, Off_Of => Stack);
           -- RAISES Constraint_Error WHEN Stack IS EMPTY.  THIS IS POOR PROGRAMMING
              PRACTICE.
        Sum           := Sum + Item;
        Number_Popped := Number_Popped + 1;
    end loop;

exception

    when Constraint_Error =>
        Put (Float (Sum) / Float (Number_Popped));

end Pop_And_Average;
```

- IN THE WORDS OF THE FAMOUS PAUL SIMON SONG, THERE MUST BE FIFTY WAYS TO RAISE Constraint_Error. HERE ARE JUST A FEW OF THEM:
  - A SUBPROGRAM CALL IN WHICH THE VALUE OF AN in OR in out ACTUAL PARAMETER IS OUTSIDE THE SUBTYPE OF THE FORMAL PARAMETER.
  - RETURNING FROM A PROCEDURE WHEN AN in out OR out FORMAL PARAMETER HOLDS A VALUE OUTSIDE THE SUBTYPE OF THE ACTUAL PARAMETER.
  - RETURNING FROM A FUNCTION WITH A VALUE OUTSIDE THE FUNCTION'S RESULT SUBTYPE.
  - A QUALIFIED EXPRESSION IN WHICH THE EXPRESSION'S VALUE IS OUTSIDE THE TYPE SPECIFIED BY THE TYPEMARK.
  - AN ASSIGNMENT IN WHICH THE VALUE ASSIGNED IS OUTSIDE THE SUBTYPE OF THE VARIABLE BEING ASSIGNED TO.
  - AN ATTEMPT TO REFER TO AN ALLOCATED VARIABLE "POINTED TO" BY THE ACCESS VALUE null.
  - AN OUT-OF-BOUNDS ARRAY INDEX VALUE.
  - A REFERENCE TO AN INDEX COMPONENT IN A CURRENTLY INACTIVE VARIANT.

PROBLEM WITH THIS APPROACH

- Constraint_Error CAN BE RAISED FOR MANY DIFFERENT REASONS, INCLUDING:
  - CALLING Pop WHEN Stack IS EMPTY
  - A POSSIBLE UNDISCOVERED PROGRAMMING ERROR IN Pop
  - THE ASSIGNMENT Sum := Sum + Item; COMPUTING A SUM OUTSIDE THE RANGE 0 .. 10_000
- THE HANDLER MAY BE INVOKED FOR THE WRONG REASON, AND NOTHING WILL APPEAR TO BE AMISS.
- THE EXCEPTION REFLECTS THE IMPLEMENTATION OF Pop (INDEXING INTO AN ARRAY) RATHER THAN THE ABSTRACT PROBLEM (TRYING TO Pop AN EMPTY STACK).

RATIONALE FOR GUIDELINE 1:

SOFTWARE SHOULD BE ROBUST. THAT IS, IT SHOULD BEHAVE SENSIBLY EVEN WHEN INVOKED IMPROPERLY. GUIDELINE 1 ENSURES THAT THE BEHAVIOR OF THE SUBPROGRAM WILL BE WELL-DEFINED FOR ALL POSSIBLE CALLS. IT ALSO HELPS ENSURE THAT ERRORS WILL NOT BE OVERLOOKED.

RATIONALE FOR GUIDELINE 2:

IT SHOULD BE DIFFICULT TO MISTAKE THE SOURCE OF THE EXCEPTION, SO THAT A HANDLER IS INVOKED FOR THE WRONG REASON.

RATIONALE FOR GUIDELINE 3:

THE IMPLEMENTATION OF THE SUBPROGRAM SHOULD BE HIDDEN FROM THE USER.

SOME DESIGN GUIDELINES

1. FOR ANY CALL, A SUBPROGRAM SHOULD EITHER COMPUTE A MEANINGFUL RESULT OR RAISE AN EXCEPTION.

   (<u>COUNTEREXAMPLE</u>: A VERSION OF Push THAT REPLACES THE TOP STACK ELEMENT WHEN CALLED WITH A FULL STACK, AND DOES NOT ALERT THE CALLER.)

2. THE EXCEPTION RAISED SHOULD BE A PROGRAMMER-DEFINED EXCEPTION WITH A NARROWLY-DEFINED MEANING.

   (<u>EXAMPLE</u>: ONE EXCEPTION FOR TRYING TO PUSH ONTO A FULL STACK, ANOTHER FOR TRYING TO POP OFF AN EMPTY STACK.

3. THE EXCEPTION SHOULD DESCRIBE THE ABSTRACT REASON FOR THE PROBLEM.

   (<u>EXAMPLES</u>: Full_Stack_Error, Empty_Stack_Error.
   <u>COUNTEREXAMPLE</u>: Stack_Array_Indexing_Error.)

INSTRUCTOR NOTES

THE PROBLEM IS THAT THE EXCEPTIONS ARE DECLARED INSIDE THE SUBPROGRAMS, SO THEY ARE ONLY VISIBLE INSIDE THE SUBPROGRAMS. THOUGH Full_Stack_Error AND Empty_Stack_Error WILL BE PROPAGATED TO THE PLACE WHERE Push AND Pop ARE CALLED, THERE WILL BE NO WAY TO NAME THE EXCEPTIONS AT THAT POINT.

THE EXCEPTIONS CAN THEREFORE ONLY BE HANDLED BY A when others HANDLER. THIS DEFEATS THE POINT OF DECLARING NARROWLY-DEFINED EXCEPTIONS.

WHAT'S WRONG WITH THIS?

```
package Stack_Package is

   type Stack_Type is limited private;
   procedure Push (Item : in Integer;  Onto   : in out Stack_Type);
   procedure Pop  (Item : out Integer; Off_Of : in out Stack_Type);

private

   type Integer_List_Type is array (1 .. 100) of Integer;
   type Stack_Type is
       record
          Top_Part      : Integer range 0 .. 100 := 0; -- Initially empty
          Contents_Part : Integer_List_Type;
       end record;

end Stack_Package;

package body Stack_Package is

   procedure Push (Item : in Integer; Onto : in out Stack_Type) is
       Full_Stack_Error : exception;
   begin
       if Onto.Top_Part < Onto.Contents_Part'Last then
          Onto.Top_Part := Onto.Top_Part + 1;
          Onto.Contents_Part (Onto.Top_Part) := Item;
       else
          raise Full_Stack_Error;
       end if;
   end Push;

   procedure Pop (Item : out Integer; Off_Of : in out Stack_Type is
       Empty_Stack_Error : exception;
   begin
       if Onto.Top_Part > 0 then
          Item := Off_Of.Contents_Part (Off_Of.Top_Part);
          Off_Of.Top_Part := Off_Of.Top_Part - 1;
       else
          raise Empty_Stack_Error;
       end if;
   end Pop;

end Stack_Package;
```

INSTRUCTOR NOTES

RATIONALE FOR GUIDELINE 1:

THE EXCEPTIONS RAISED BY A SUBPROGRAM ARE LOGICALLY PART OF A SUBPROGRAM'S INTERFACE. IF A SUBPROGRAM IS PART OF A PACKAGE'S INTERFACE, THEN SO ARE THE EXCEPTIONS RAISED BY THAT SUBPROGRAM.

RATIONALE FOR GUIDELINE 2:

THE SUBPROGRAM AND EXCEPTION MUST BE PROVIDED TOGETHER, BECAUSE NEITHER IS MEANINGFUL WITHOUT THE OTHER.

MORE DESIGN GUIDELINES

1. WHEN A PACKAGE PROVIDES SUBPROGRAMS, THE EXCEPTIONS PROVIDED BY THE SUBPROGRAMS SHOULD BE DECLARED IN THE PACKAGE SPECIFICATION.

```
package Stack_Package is

   type Stack_Type is limited private;

   procedure Push (Item : in Integer; Onto : in out Stack_Type);
   procedure Pop  (Item : out Integer; Off_Of : in out Stack_Type);

   Full_Stack_Error : exception;
      -- RAISED BY Push WHEN CALLED WITH A FULL STACK.

   Empty_Stack Error : exception;
      -- RAISED BY Pop WHEN CALLED WITH AN EMPTY STACK.

private

   type Integer_List_Type is array (1 .. 100) of Integer;
   type Stack_Type is
      record
         Top_Part      : Integer range 0 .. 100 := 0;   -- Initially empty
         Contents_Part : Integer_List_Type;
      end record;

end Stack_Package;
```

2. WHEN A STANDALONE SUBPROGRAM IS TO RAISE AN EXCEPTION, THE SUBPROGRAM SHOULD BE PLACED IN A PACKAGE THAT PROVIDES BOTH THE SUBPROGRAM AND THE EXCEPTION.

```
with Stack_Package;

package Pop_And_Average_Package is
   procedure Pop_And_Average (Stack : in out Stack_Package.Stack_Type);
   Averaging_Error : exception;
      -- RAISED BY A CALL ON Pop_And_Average WITH AN EMPTY STACK.
end Pop_And_Average_Package;
```

INSTRUCTOR NOTES

THE DIFFERENCE IS THAT THE VERSION OF Stack_Package ON THE RIGHT PROVIDES A FUNCTION Is_Empty THAT ALLOWS US TO DETERMINE WHETHER A STACK IS EMPTY WITHOUT ACTUALLY RAISING AN EXCEPTION.

THE VERSION ON THE RIGHT SHOULD BE CLEARER TO MOST STUDENTS, BECAUSE THE FLOW OF CONTROL IS MORE EXPLICIT. FAMILIAR CONTROL STRUCTURES ARE USED AND THE CONDITIONS UNDER WHICH THE LOOP IS EXITED ARE OBVIOUS.

STUDENTS WHO BELIEVE THAT THE VERSION ON THE LEFT IS CLEARER MAY BE EXCUSED FROM THE ROOM FOR THE REMAINDER OF SECTION 9. WHEN THEY RETURN, THEY SHOULD BE WARNED THAT THEY ARE ON PROBATION.

WHICH IS CLEARER?

```
package Stack_Package is
    type Stack_Type is limited private;
    procedure Push (Item : in Integer; Onto : in out Stack_Type);
    procedure Pop (Item : out Integer; Off_Of : in out Stack_Type);
    Full_Stack_Error, Empty_Stack_Error : exception;
private
    ...
end Stack_Package;

with Stack_Package;

package Pop_And_Average_Package is
    procedure Pop_And_Average (Stack : in out Stack_Package.Stack_Type);
    Averaging_Error : exception;
end Pop_And_Average_Package;

with Text_IO;

package body Pop_And_Average_Package is
    procedure Pop_And_Average (Stack : in out Stack_Package.Stack_Type) is
       package Type_Float_IO is new Text_IO.Float_IO (Float);
       use Type_Float_IO;
       Number_Popped : Integer := 0;
       Sum           : Integer range 0 .. 10_000 := 0;
       Item          : Integer;
    begin
       loop
          Stack_Package.Pop (Item, Off_Of => Stack);
          Sum := Sum + Item;
          Number_Popped := Number_Popped + 1;
       end loop;
    exception
       when Stack_Package.Empty_Stack_Error =>
          if Number_Popped > 0 then
             Put (Float (Sum) / Float (Number_Popped));
          else
             raise Averaging_Error;
          end if;
    end Pop_And_Average;
end Pop_And_Average_Package;
```

```
package Stack_Package is
    type Stack_Type is limited private;
    procedure Push (Item : in Integer; Onto : in out Stack_Type);
    procedure Pop (Item : out Integer; Off_Of : in out Stack_Type);
    function Is_Full (Stack : Stack_Type) return Boolean;
    function Is_Empty (Stack : Stack_Type) return Boolean;
    Full_Stack_Error, Empty_Stack_Error : exception;
private
    ...
end Stack_Package;

with Stack_Package;

package Pop_And_Average_Package is
    procedure Pop_And_Average (Stack : in out Stack_Package.Stack_Type);
    Averaging_Error : exception;
end Pop_And_Average_Package;

with Text_IO;

package body Pop_And_Average_Package is
    procedure Pop_And_Average (Stack : in out Stack_Package.Stack_Type) is
       package Type_Float_IO is new Text_IO.Float_IO (Float);
       use Type_Float_IO;
       Number_Popped : Integer := 0;
       Sum           : Integer range 0 .. 10_000 := 0;
       Item          : Integer;
    begin
       while not Stack_Package.Is_Empty (Stack) loop
          Stack_Package.Pop (Item, Off_Of => Stack);
          Sum := Sum + Item;
          Number_Popped := Number_Popped + 1;
       end loop;
       if Number_Popped > 0 then
          Put (Float (Sum) / Float (Number_Popped));
       else
          raise Averaging_Error;
       end if;
    end Pop_And_Average;
end Pop_And_Average_Package;
```

INSTRUCTOR NOTES

RATIONALE:

BASING FLOW OF CONTROL ON THE RAISING OF EXCEPTIONS IS USUALLY LESS CLEAR, AS SHOWN ON THE PREVIOUS SLIDE.

EXCEPTIONS CAN BE RESERVED FOR ABNORMAL SITUATIONS, WITH THE PACKAGE USER GIVEN A FAIR OPPORTUNITY TO AVOID EXCEPTIONS BY USING THE PACKAGE PROPERLY.

IT IS NOT ALWAYS PRACTICAL TO FOLLOW THIS GUIDELINE. A CONSIDERABLE AMOUNT OF COMPUTATION MAY BE NECESSARY TO DETERMINE WHETHER AN EXCEPTION SHOULD BE RAISED, AND THIS MAY ENTAIL MOST OF THE WORK OF THE SUBPROGRAM THAT ACTUALLY RAISES THE EXCEPTION. CERTAIN EXCEPTIONS, LIKE Storage_Error AND THE I/O EXCEPTIONS Device_Error AND Data_Error CANNOT EASILY BE PREDICTED. SIMILARLY, THE ONLY WAY TO TELL WHETHER A FILE EXISTS BEFORE OPENING IT IS BY TRYING TO OPEN IT AND SEEING WHETHER Name_Error IS RAISED. (IN A MULTIUSER FILE SYSTEM, A BOOLEAN FUNCTION File_Exists WOULD BE USELESS ANYWAY, SINCE ANOTHER USER COULD DELETE A FILE BETWEEN THE TIME THIS FUNCTION RETURNED True AND THE TIME THE FILE WAS ACTUALLY OPENED.)

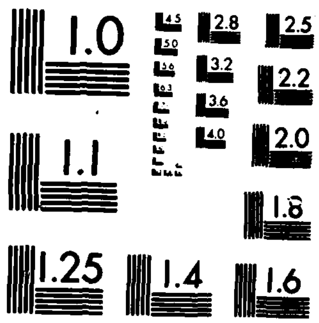
A FINAL GUIDELINE

WHENEVER FEASIBLE, A PACKAGE PROVIDING SUBPROGRAMS THAT RAISE EXCEPTIONS SHOULD ALSO PROVIDE FUNCTIONS TO DETERMINE, WITHOUT ACTUALLY RAISING AN EXCEPTION, WHETHER CALLS ON THOSE SUBPROGRAMS WOULD RAISE EXCEPTIONS.

```
with Stack_Package;

package Pop_And_Average_Package is
   procedure Pop_And_Average (Stack : in out Stack_Package.Stack_Type);
   function Unaverageable (Stack : Stack_Type) return Boolean
      renames Stack_Package.Is_Empty;

   Averaging_Error : exception;

end Pop_And_Average_Package;
```

AD-A165 075 ADA (TRADEMARK) TRAINING CURRICULUM: ADVANCED ADA TOPICS L305 TEACHER'S GUIDE VOLUME 1(U) SOFTECH INC WALTHAM MA 1986 DAAB07-83-C-K506 5/5

UNCLASSIFIED F/G 9/2 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

INSTRUCTOR NOTES

AMONG THE ISSUES COVERED IN THE CASE STUDY BUT NOT HERE ARE:

- CONTROL STRUCTURES BASED ON EXCEPTION HANDLING
- GUIDELINES FOR HANDLING EXCEPTIONS
- THE ROLE OF THE PROGRAMMER VERSUS THE ROLE OF THE DESIGNER VIS-A-VIS EXCEPTIONS

STRATEGIES FOR ERROR RECOVERY AND FAULT-TOLERANT COMPUTING ARE A COMPLEX SYSTEM DESIGN ISSUE BEYOND THE SCOPE OF THIS MODULE.

ANOTHER ISSUE, EXCEPTIONAL SITUATIONS IN MULTITASK ENVIRONMENTS, IS DEALT WITH IN L401, IN THE SECTION ON MONITORS.

SUMMARY

1. FOR ANY CALL, A SUBPROGRAM SHOULD EITHER COMPUTE A MEANINGFUL RESULT OR RAISE AN EXCEPTION.

2. THE EXCEPTION RAISED SHOULD BE A PROGRAMMER-DEFINED EXCEPTION WITH A NARROWLY-DEFINED MEANING.

3. THE EXCEPTION SHOULD DESCRIBE THE ABSTRACT REASON FOR THE PROBLEM.

4. WHEN A PACKAGE PROVIDES SUBPROGRAMS, THE EXCEPTIONS PROVIDED BY THE SUBPROGRAMS SHOULD BE DECLARED IN THE PACKAGE SPECIFICATION.

5. WHEN A STANDALONE SUBPROGRAM IS TO RAISE AN EXCEPTION, THE SUBPROGRAM SHOULD BE PLACED IN A PACKAGE THAT PROVIDES BOTH THE SUBPROGRAM AND THE EXCEPTION.

6. WHENEVER FEASIBLE, A PACKAGE PROVIDING SUBPROGRAMS THAT RAISE EXCEPTIONS SHOULD ALSO PROVIDE FUNCTIONS TO DETERMINE, WITHOUT ACTUALLY RAISING AN EXCEPTION, WHETHER CALLS ON THOSE SUBPROGRAMS WOULD RAISE EXCEPTIONS.

A FURTHER DISCUSSION OF EXCEPTIONS IS FOUND IN CASE STUDY 2.4.1, "USE OF EXCEPTIONS," IN Ada CASE STUDIES II.

Material: Advanced Ada Topics (L305), Volume I

We would appreciate your comments on this material and would like you to complete this brief questionaire. The completed questionaire should be forwarded to the address on the back of this page. Thank you in advance for your time and effort.

1. Your name, company or affiliation, address and phone number.

2. Was the material accurate and technically correct?

   Yes ☐ No ☐

   Comments:

3. Were there any typographical errors?

   Yes ☐ No ☐

   If yes, on what pages?

4. Was the material organized and presented appropriately for your applications?

   Yes ☐ No ☐

   Comments:

5. General Comments:

place
stamp
here

COMMANDER
US ARMY MATERIEL COMMAND
ATTN: AMCDE-SB (OGLESBY)
5001 EISENHOWER AVENUE
ALEXANDRIA, VIRGINIA 22233

END
FILMED
4-86
DTIC